TRABAJO FIN DE GRADO

INGENIERÍA EN INFORMÁTICA

# Buscando soluciones del Cubo de Rubik


**Autor**
Jorge Castillo Matas

**Tutor**
Jesús García Miranda



Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación


—

Granada, sepetiembre de 2019

## Título del Proyecto: Buscando soluciones al Cubo de Rubik

Jorge Castillo Matas

**Palabras clave**: Cubo de Rubik, Metaheurísticas, Algoritmos Genéticos, Thistlethwaite

### Resumen

Este proyecto consiste en la búsqueda de soluciones al Cubo de Rubik mediante el uso de Metaheurísticas. Por el hecho de que hay pocos antecedentes en el empleo de este tipo de algoritmos a este problema, se empieza intentándolo con un cubo más pequeño como es el 2x2x2. Después de comprobar que las Metaheurísticas funcionan para este rompecabezas se comienza su estudio en el 3x3x3. Este, habrá que dividirlo en pasos, ya que encontrar una solución sin particionamiento es un proceso muy complejo. Para ello, nos apoyaremos en el algoritmo Thistlethwaite que nos proporcionará ciertas facilidades a la hora de encontrar soluciones.

## Project Title: Looking for Rubik's Cube solutions

Jorge Castillo Matas

**Keywords**: Rubik's Cube, Metaheuristics, Genetics Algorithms, Thistlethwaite

### Abstract

This project consists in looking for Rubik's Cube solutions by using Metaheuristics. By the fact that there are few jobs using this kind of algorithm applied to this problem, we started trying for a smaller cube as 2x2x2. After checking that Metaheuristics works for this puzzle we start the study on 3x3x3. The Rubik's Cube will be splited into steps, since finding an undivided solution is a very complex process. For it, we will rely on the Thistlethwaite algorithm that will be benefit us with facilities in the process of finding solution.

Yo, **Jorge Castillo Matas**, alumno de la titulación Graduado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 25613145T, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jorge Castillo Matas

Granada a 5 de septiembre de 2019.

D. **Jesús García Miranda**, Profesor del Área de Álgebra del Departamento de Álgebra de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado ***Buscando soluciones al Cubo de Rubik***, ha sido realizado bajo su supervisión por **Jorge Castillo Matas**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 5 de septiembre de 2019.

**El tutor:**

**Jesús García Miranda**

# Agradecimientos

Agradezco a Jesús García Miranda su atención y ayuda durante la realización de este proyecto.

# Índice general

# Capítulo 1

# Introducción

Resolver el Cubo de Rubik parece una labor bastante compleja y que requiere de cierta paciencia. Esto es así para un tanto por ciento bastante alto de las personas. Internet y sobretodo Youtube, cambió ese pensamiento y cada vez más y más gente sabe resolver el «dichoso» puzzle de 6 caras. Los métodos para resolverlo van creciendo desde que este salió a la venta. Y en esta ocupación de desarrollar métodos para el Cubo de Rubik han participado matemáticos e informáticos en su gran mayoria. Hace ya más de 45 años que se inventó y desde entonces se ha invertido mucho tiempo de estudio en él. Hay bastantes programas informáticos que dada una posición del cubo, te muestran una secuencia de movimientos que lo resuelve. Esta secuencia suele ser bastante corta teniendo en cuenta que el número de Dios es 20 para este puzzle. Esto quiere decir que, desde cualquier configuración de este rompecabezas hay un mínimo de movimientos para resolverla que es como máximo de 20 movimientos. Estos programas suelen usar IDA* (Iterative Deepning A*) para encontrar la solución. Es decir, parten del estado inicial el algoritmo va generando un grafo en el que las transiciones son movimientos del cubo y los nodos son estados del cubo. Este algoritmo explora los estados más prometedores antes que los demás para ahorrar tiempo de búsqueda. Esta forma de solucionar el Cubo de Rubik es muy lógica. De hecho, cuando nos «enfrentamos» con este puzzle, sin usar computadoras, una buena estrategia es la de ir haciendo y deshaciendo movimientos hasta quedarnos con una configuración del cubo que nos guste más que la que teniamos de inicio. Otra similitud puede ser otro rompecabezas en el sentido más estricto de la palabra, el ajedrez. En el ajedrez, dada una posición, la manera inteligente de actuar es, ir probando «todos» los movimientos posibles y sus sucesiones hasta ver cual nos conduce al estado más satisfactorio. Por lo tanto, ante este tipo de juegos que nos obligan a pensar, esta táctica parece bastante usada.

Pero hace un tiempo me encontré con una asignatura de nombre Metaheurísticas que me llamó mucho la atención. Nunca me había imaginado

una metodología para resolver problemas tan visual y tan «computacional». Y yo, como gran apasionado a este rompecabezas que fui, pensé en si estos algoritmos podrían enfrentarse al Cubo de Rubik. A la mayoría de las personas que llevan poco tiempo usando metaheurísticas no creen que puedan funcionar tan bien para problemas tan diversos. Y esto mismo me pasó a mi, pensé que podría plantearlo como TFG pero que seguramente no se podría. Y hoy, ya tengo respuesta a esta duda y me alegro de que la respuesta sea positiva.

## 1.1. Objetivos

Los objetivos principales de este proyecto son:

- Resolver el 2x2x2 usando Metaheurísticas y sin dividirlo en pasos.
- Solucionar el 3x3x3 usando Metaheurísticas, aunque para ello tengamos que dividirlo en fases.
- Usar Algoritmos Evolutivos, como pueden ser los Algoritmos Genéticos, para la consecución del objetivo anterior e intentar aplicar la metaheurística de la Colonia de Hormigas a este problema.
- No aportarle a los algoritmos secuencias preestablecidas que les sirvan de ayuda. Estas Metaheurísticas solo trabajarán con los movimientos individuales que se pueden aplicar al puzzle y sus consecuencias en este.

# Capítulo 2

# Antecedentes

El problema de encontrar la solución del Cubo de Rubik usando Metaheurísticas no se ha estudiado demasiado. Pero hay ciertos trabajos y proyectos que me han servido para seleccionar planteamientos y para marcarme objetivos. En todos ellos usan Algoritmos Genéticos para su resolución y en uno de ellos el Algoritmo Genético comparte espacio con el Enfriamiento Simulado.

Empezaré por el proyecto que peores resultados ha obtenido, puesto que tenía un objetivo inicial demasiado ambicioso. Se trata de un proyecto realizado en Swift que perseguía la posibilidad de encontrar una secuencia de movimientos que resolviese el Cubo de Rubik sin dividirlo en fases. Como ya he dicho antes, no consiguió solucionar el problema, aunque se quedó a solo 4 piezas de las 26 totales tras 877 minutos de ejecución. Continúo con el que mejores resultados ha obtenido usando Metaheurísticas. En este proyecto, se usan Algoritmos Genéticos y Enfriamiento Simulado. Con ambos algoritmos consigue resolver el 3x3x3 sin dividirlo en etapas. Además, los resultados conseguidos para el Algoritmo Genético son muy buenos, puesto que encuentra una solución de 22 movimientos el 43 % de las veces. El algoritmo de Enfriamiento Simulado no funciona tan bien, lo consigue en unos 219 movimientos de media, pero tarda menos tiempo.

El resto de trabajos dividen el Cubo de Rubik en fases para facilitar el cometido. El primero proyecto de este grupo lo particiona en 4 pasos. Y consigue solucionar el rompecabezas en 50 movimientos de media. El siguiente trabajo también lo fracciona en 4 etapas y consigue unos resultados de entre 30 y 38 movimientos. Por último, se consigue resolver en una media de 31 movimientos partiendo la solución en dos etapas.

# Capítulo 3

# Metaheurísticas

## 3.1. Definición

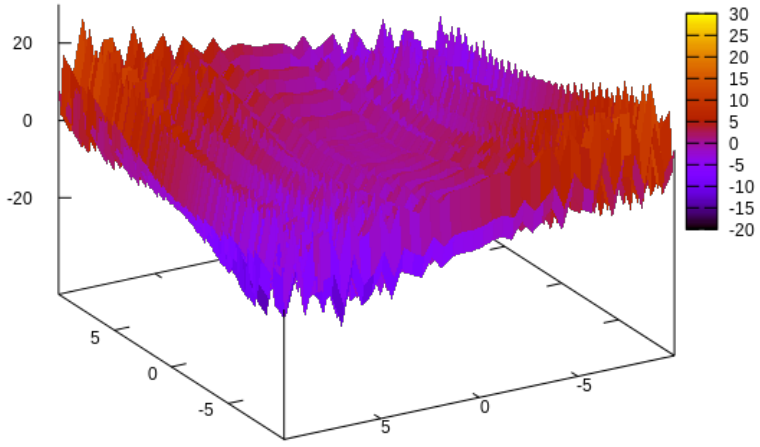
Los algoritmos metaheurísticos son algoritmos aproximados de optimización y búsqueda de proposito general. Normalmente, se aplican a problemas que no tienen un algoritmo o heurística especifica que dé una solución satisfactoria, o bien cuando no es posible implementar ese método óptimo. La mayoría de metaheurísticas tienen como objetivo los problemas de optimización combinatoria. Pero se pueden aplicar a cualquier problema que se pueda reformular en terminos heurísticos.

Para entender mejor como funcionan las metaheurísticas tenemos que saber que es una heurística. La heurística es un procedimiento basado en la experiencia que puede usarse como ayuda para la resolución de problemas. Dicha heurística aplicada a una solución nos valorará la calidad de dicha solución y este valor será el **valor heurístico**. Un ejemplo muy simple y con el que queda muy claro es con el Problema del Viajante (TSP). Dada una lista de ciudades y las distancias entre cada par de ellas. ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?. Para este ejemplo, si tenemos 8 ciudades una posible solución es:

$$(1, 8, 2, 6, 3, 5, 4, 7)$$

Si visitamos las ciudades en ese orden, recorreremos una distancia $D$ y esto será la heurística del problema. Cuanto menor sea el valor heurístico $D$ de la solución, mejor será.

El objetivo de las metaheurísticas es el de encontrar una buena solución. Imaginémos un espacio de soluciones dictaminado por la siguiente imagen.




En la que las soluciones estan representadas sobre el eje Y. El valor heurístico de cada solución es igual al valor de la coordenada Y de dicho punto en el espacio. Las soluciones del gráfico están posicionadas de acuerdo a la «mutación» que definamos. Con mutación me refiero a variación de la solución para obtener una nueva solución. En el TSP, si definimos que nuestra mutación se basa en el intercambio de dos ciudades:

$$S1 : (1, \mathbf{2}, 3, 4, 5, \mathbf{6}, 7, 8)$$

$$S2 : (1, \mathbf{6}, 3, 4, 5, \mathbf{2}, 7, 8)$$

$S1$ y $S2$ serían «vecinas» en el espacio de búsqueda. Si definimos otra mutación el entorno de cada solución variará, y por lo tanto, nuestro espacio de búsqueda se distribuirá de diferente manera. Teniendo ya un espacio de búsqueda definido podremos aplicar nuestra metaheurística. Esta, intentará encontrar el mínimo (o máximo en algunos casos) valor heurístico mediante su desplazamiento, cruzamiento,... y otras técnicas usadas. Supongamos que el espacio de soluciones es un terreno con ciertas montañas. Las soluciones peores estarán en la cima de las montañas y las mejores estarán en los puntos de menor altitud. ¿Porqué las soluciones se distrubuyen de manera que parezca un terreno montañoso?. Porque las soluciones proximas serán muy parecidas y por lo tanto, su valor heurístico será similar. Recordemos que el espacio de soluciones se distribuye de acuerdo a nuestro «operador de vecino», que antes hemos llamado mutación. Un operador de vecino, dada

una solución de entrada, nos retornará una solución vecina y por lo tanto de parecido valor heurístico.

Entonces, la metaheurística podría ser cualquier mecanismo que tras ciertos movimientos sobre el espacio de búsqueda encontrase un punto de baja altitud. Cuanto menor valor encuentre dicha metaheurística, mejor adaptación a dicho problema tendrá.

## 3.2. Aspectos importantes a tener en cuenta

### 3.2.1. Representación de la solución

Debemos encontrar una forma de representar la solución que cumpla las siguientes características:

- **Facilidad al realizar mutaciones**: facilitar las mutaciones en la representación nos puede ahorrar mucho tiempo.
- **Simplicidad**: nos ayudará en el uso de memoria.
- **Proceso de cálculo del valor heurístico**: dado que vamos a tener que calcular el valor heurístico de las soluciones constantemente, hay que disminuir el tiempo de dicho cálculo. Distintas representaciones nos ayudarán o perjudicarán en esta faceta.

### 3.2.2. Heurística

Desarrollar una buena heurística le servirá de mucha ayuda al algoritmo metaheurístico dado que las metaheurísticas se nutren de la calidad de las soluciones ,y a partir de ahí, actuan de una forma u otra. Pero como explico en el siguiente punto, hay que tener en cuenta lo costoso, computacionalmente hablando, que es calcular dicho valor heurístico. Entonces, estamos en un terreno algo inestable y abstracto en el que cualquier metaheurística muy simple puede sorprendernos con sus resultados. Y también podemos implentar una metaheurística muy compleja con la que no obtengamos resultados beneficiosos.

### 3.2.3. Función objetivo

La función objetivo se encargará de calcular el valor heurístico de una solución. Como hemos mencionado antes, este proceso se ejecutará muchas veces en la optimización usando metaheurísticas. Por lo tanto, es vital que sea rápido en su ejecución, porque lo que tarde la ejecución del algoritmo va a depender en gran medida de esta función.

En el ejemplo del TSP, la función objetivo se encargaría de buscar en la matriz de distancias las longitudes de trayecto entre las ciudades adyacentes en el camino y las sumará. Con lo cual, optimizar el proceso de búsqueda

de distancias sería una tarea pendiente a la hora de optimizar la función objetivo. La **paralelización** de código en esta función es otra de las mejoras que nos puede ayudar para ciertos problemas.

### 3.2.4. Generación de vecinos

Para la busqueda de una mejor solución vamos a tener que ir desplazándonos por el espacio de búsqueda. Las mutaciones van a ser la forma que tengamos de movernos. Por eso, es importante que las soluciones vecinas tengan un valor heurístico parecido a las originales para que nuestro espacio de búsqueda tenga un aspecto parecido al de muchas «colinas», donde tenemos que ir bajando por estas hasta llegar a una solución de calidad.

Aquí, nos surge otro problema que hay que comentar, el de los **óptimos locales**. Cuando bajamos por una colina en busca de una solución, podemos encontrarnos con un punto en el que su entorno sea peor que él, esto es un óptimo local. Es impresindible que la metaheurística tenga una forma de salir de este óptimo local para explorar por otra parte del espacio de soluciones. Pero esto ya lo veremos más adelante.

## 3.3. Clasificación de las metaheurísticas

Podemos clasificar las metaheurísticas en tres tipos distintos:

- **Basadas en trayectorias**: parten de un punto y van avanzando poco a poco hacia una mejor solución. Por eso se llaman así, siguen una trayectoría hasta su objetivo. Algunas metaheurísticas de este tipo como el enfriamiento simulado y la búsqueda Tabú permitan movimientos de empeoramiento cuando llegan o se acercan a un óptimo local. Otras, ante esta situación, empiezan la búsqueda partiendo desde otro punto (Busquedas Multiarranque, ILS, VSN,...).

- **Basadas en poblaciones**: está compuesta por modelos de evolución basados en poblaciones cuyos elementos representan soluciones a problemas. Estos elementos van a ser capaces de reproducirse mediante la combinación de dos de ellos y también de mutar por sí mismos. Las mejores soluciones perdurarán sobre las generaciones y las peores se eliminarán (proceso de selección). Los algoritmos genéticos, algoritmos meméticos, evolución diferencial,... pertenecen a este tipo.

- **Basadas en adaptación social**: son algoritmos o mecanismos distribuidos de resolución de problemas inspirados en el comportamiento colectivo de sociedades de animales. Pueden estar basados en las colonias de hormigas, en las nubes de párticulas,...

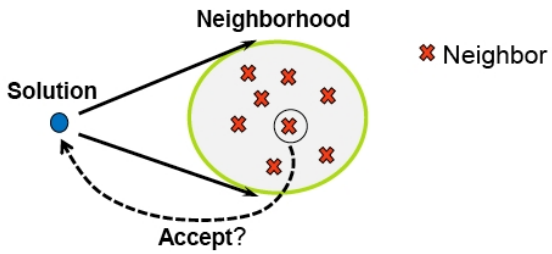
# Capítulo 4

# Metaheurísticas usadas en este proyecto

## 4.1. Búsqueda Local Multiarranque

Para entender la Búsqueda Local Multiarranque hay que comprender primero lo que es la Búsqueda Local. La Búsqueda Local es un proceso que, dada la solución actual, selecciona iterativamente una solución de su entorno para continuar la búsqueda. Es decir, generamos los vecinos de la solución actual hasta obtener una solución mejor, hasta obtener la mejor solución o hasta explorar todo el vecindario y ver que no hay una solución mejor. Aquí radica el gran problema de la Búsqueda Local, que cae en óptimos locales muy rápidamente. De esta forma, tendríamos tres tareas para implementar este algoritmo:

- Fijar una codificación para las soluciones.
- Definir un operador de generación de vecino y, en consecuencia, se fija una estructura de entorno para las mismas.
- Escoger una solución del entorno de la solución actual hasta que se satisfaga el criterio de parada.




http://paradiseo.gforge.inria.fr/pub/img/schemaLS.jpg

Veamos el pseudocódigo de este algoritmo:

$GENERA(SoluciónIncial)$;
$SoluciónActual \leftarrow SoluciónInicial$;
$MejorSolución \leftarrow SoluciónActual$;
**while** *Criterio de parada* **do**
  $SoluciónVecina \leftarrow GENERAVECINO(SoluciónActual)$;
  **if** *Acepta(Solución Vecina)* **then**
    $SoluciónActual \leftarrow SoluciónVecina$;
  **end**
  **if** $FUNOBJ(SoluciónActual) < FUNOBJ(MejorSolución)$ **then**
    $mejorSolución \leftarrow SoluciónActual$;
  **end**
**end**
**return** $MejorSolución$;

**Algorithm 1:** Búsqueda Local

Siendo las funciones:

- GENERA: la que genera una solución inicial.
- GENERAVECINO: la que genera un vecino de la solución que se le pasa como parámetro.
- FUNOBJ: la que calcula el valor heurístico de la solución pasada como parámetro.

Hay dos variantes de la Búsqueda Local:

- del Mejor: que consiste en explorar todo el entorno y escoger la mejor en caso de mejorar a la actual. Esta búsqueda es más reconmendable en problemas con el espacio de soluciones «regular». Es decir, siendo el valor heurístico de los vecinos bastante parecido a los de la solución original.
- del Primer Mejor: se va generando paso a paso el entorno de la solución actual hasta que se obtiene una solución vecina que mejora a la actual o se construye el entorno completo.

Ya podemos definir la Búsqueda Local con Arranque Multiple con mayor facilidad. Se trata de un algoritmo de búsqueda global que itera las dos estapas siguientes:

1. Generación de una solución inicial $S$.

2. Búsqueda Local sobre $S$ que nos devolvería $S'$.

Estos pasos se repiten hasta que se satisfaga algún criterio de parada. Después se devuelve como salida del algoritmo la solución $S'$ que mejor valor de la función objetivo presente. En la etapa 1 se pueden generar soluciones inciales aleatorias o partir de una solución de mayor calidad usando un algoritmo Greedy. Esto nos puede ahorrar iteraciones para ciertos problemas. Y en la etapa 2 se puede usar Búsqueda Local o se puede usar Enfriamiento Simulado,... o cualquier algoritmo basado en trayectorias. En cuanto a la condición de parada, se han propuesto desde criterios simples, como el de parar después de un número dado de iteraciones, hasta criterios que analizan la evolución de la búsqueda. Con la búsqueda Local con Arranque Multiple evitamos caer en óptimos locales y no poder salir de ellos puesto que al estar en un óptimo local comenzamos la búsqueda desde otro punto.

$S_{act} \leftarrow GeneraSolución()$;
$MejorSolución \leftarrow S_{act}$;
**while** *Criterio de parada* **do**
  $S' \leftarrow BUSQUEDALOCAL(S_{act})$;
  **if** $S' < MejorSolución$ **then**
    $MejorSolución \leftarrow S'$;
  **end**
  $S_{act} \leftarrow GeneraSolucion()$;
**end**
**return** $MejorSolución$;

**Algorithm 2:** Búsqueda Local Multiarranque

## 4.2. Algoritmos Genéticos

La computación evolutiva está compuesta por modelos basados en poblaciones cuyos elementos representan soluciones a problemas. La simulación de este proceso en un ordenador resulta ser una técnica de optimización probabilística, que con frecuencia mejora a otros métodos clásicos en problemas difíciles. En la evolución natural, una entidad o individuo tiene la habilidad de reproducirse y hay una población de tales individuos con dicha cualidad. Existe alguna variedad, diferencia, entre los individuos que se reproducen. Algunas diferencias en la habilidad para sobrevivir en el entorno están asociadas con esa variedad. Los mecanismos que conducen esta evolución no

son totalmente conocidos, pero sí algunas de sus características, que son ampliamente aceptadas:

1. La evolución es un proceso que opera sobre los cromosomas más que sobre las estructuraas de la vida que están codificadas en ellos.

2. La selección natural es el enlace entre los cromosomas y la actuación de sus estrucuras decodificadas.

3. El proceso de reproducción es el punto en el cual la evolución toma parte, actúa.

4. La evolución biológica no tiene memoria.

Los algoritmos Genéticos son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de evolución natural y evolución genética. Estos funcionan entre el conjunto de soluciones de un problema llamado fenotipo, y el conjunto de individuos de una **población** natural, codificando la información de cada solución en una cadena, llamada **cromosoma**. Los símbolos que conforman el cromosoma son llamados **genes**. Los cromosomas evolucionan a traves de iteraciones, llamadas **generaciones**. En cada generación los cromosomas son evaluados de acuerdo a la función objetivo. Las siguientes generaciones son generadas aplicando los operadores genéticos repetidamente, siendo estos los operadores de selección, cruzamiento, mutación y reemplazo. Los operadores de selección realizan la función de elegir los cromosomas que formarán la siguiente generación. Puesto que, tras realizar cruzamiento de cromosomas tendremos una cantidad de individuos superior al tamaño de población que queremos. El operador de cruzamiento se basa en la unión de dos padres para la formación de varios hijos. Estos hijos podrán sustituir a los padres, podrán sustituir a los padres solo los que tengan mejor valor heurístico que ellos,... y aquí es donde actúa el operador de reemplazo. Las mutaciones serán aplicadas a un porcentaje de la población y consistirán en variaciones en estos individuos.

A continuación veremos su pseudocódigo:

$t \leftarrow 0$;
$InicializarPoblación(t)$;
$EvaluarPoblación(t)$;
**while** *Criterio de parada* **do**
  $t \leftarrow t + 1$;
  $SeleccionarPoblación' desde Población(t-1)$;
  $RecombinarPoblación'$;
  $MutarPoblación'$;
  $ReemplazarPoblación(t) a partir de Población(t-1) y Población'$;
  $EvaluarPoblación(t)$;
**end**

**Algorithm 3:** Algoritmo Genético

Existen dos tipos de modelos genéticos:

- **Generacional**: durante cada iteración se crea una población completa con nuevos individuos. La nueva población reemplaza directamente a la antigua.
- **Estacionario**: durante cada iteración se escogen dos padres de la población y se les aplican los operadores genéticos. Los descendientes reemplazan al mismo número de cromosomas de la población anterior.

En la estrategia de selección debemos garantizar que los mejores individuos tengan una mayor posibilidad de ser padres (reproducirse) frente a los individuos menos buenos. Debemos ser cuidadosos para dar una oportunidad de reproducirse a los individuos menos buenos. Éstos pueden incluir material genético útil en el proceso de reproducción. Esta idea nos define la **presión selectiva** que determina en qué grado la reproducción está dirigida por los mejores individuos.

Podriamos tener uno o más operadores de cruce para nuestra representación. Algunos aspectos importantes a tener en cuenta son:

- Los hijos deberían heredar algunas características de cada padre. Si éste no es el caso, entonces estamos ante un operador de mutación.
- Se debe diseñar de acuerdo a la representación.
- La recombinación debe producir cromosomas válidos.
- Se utiliza con una probabilidad alta de actuación sobre cada pareja de padres a cruzar, si no actúa los padres son descendientes del proceso de recombinación de la pareja.

También podemos tener uno o más operadores de mutación para nuestra representación. Algunos aspectos importantes a tener en cuenta son:

- Debe permitir alcanzar cualquier parte del espacio de búsqueda.

- El tamaño de la mutación debe ser controlado.
- Debe producir cromosomas válidos.
- Se aplica con una probabilidad muy baja de actuación sobre cada descendiente obtenido tras aplicar el operador de cruce.

Para la estrategia de reemplazamiento podemos utilizar métodos de reemplazamiento aleatorios, o deterministas. Podemos decidir no reemplazar al mejor cromosoma de la población: **Elitismo**. Un modelo con alto grado de elitismo consiste en utilizar una población intermedia con todos los padres y todos los descendientes y seleccionar los $N$ mejores. Esto se combina con otras componentes con alto grado de diversidad. Aunque los Algoritmos Genéticos son puramente exploradores (diversos), llegados a un número de generaciones suelen converger. Esto significa que nos queda una población uniforme y poco variada. Para evitar esto, debemos tener en cuenta algunas estrategias:

- Diversidad en la mutación para que el cromosoma final no sea del entorno del cromosoma inicial.
- Diversidad en el cruce: hay muchas formas de conseguir un cruce diverso mediante cierta aleatoriedad.
- Estrategias de reemplazamiento entre padres e hijos.
- Reinicialización de la población cuando converge.

Aunque no nos debemos olvidar de que la convergencia es necesaria también. Si usamos estas estrategias para favorecer a la diversidad debemos también usar estrategias elitistas para beneficiar a la convergencia.

## 4.3. Algoritmos Meméticos

Son algoritmos basados en la evolución de poblaciones que para realizar búsqueda heurística intenta utilizar todo el conocimiento sobre el problema (usualmente conocimiento en términos de algoritmos específicos de búsqueda local para el problema). La hibridación entre los Algoritmos Evolutivos y la Búsqueda Local cobra sentido cuando sabemos las virtudes de cada algoritmo. Los Algoritmos Evolutivos son buenos exploradores, ya que una de sus características es que poseen una población muy variada. Sin embargo, cuando tienen una buena solución no tratan de mejorarla de forma individual. Es decir, no es un algoritmo «explotador». Por otro lado, tenemos a la Búsqueda Local que sí que cumple lo último comentado, es buena explotadora. Pero su gran defecto es que apenas explora nada, solo un area pequeña de todo el espacio de soluciones. Luego, ya sabemos porqué hay que hibridar estos dos tipos de algoritmos.

En los Algoritmos Meméticos se utiliza el término de **agentes** en lugar de individuos ya que se consideran una extensión de los segundos. Tanto la selección como la actualización (reemplazo), son procesos puramente competitivos. La reproducción es la encargada de crear nuevos agentes (cooperación). Aunque puede aplicarse una gran variedad de operadores de reproducción, existen básicamente dos: recombinación y mutación. La recombinación realiza el proceso de cooperación. Crea nuevos agentes utilizando principalmente la información extraída de los agentes recombinados. La mutación permte incluir información externa creando nuevos agentes mediante modificación parcial del agente mutado. Los optimizadores locales son considerados como un operador más y pueden aplicarse de diferentes formas:

- En la fase de incialización de la población.
- En cada generación o cada cierto número de ellas.
- Como el fin del ciclo reproductivo o durante los operadores de recombinación.
- A toda la población o a un subconjunto de ella.

Al aplicar los optimizadores locales, es esencial regular adecuadamente el equilibrio entre anchura y profundidad. Siendo la achura la frecuencia de aplicación del optimizador y la profundidad la intensidad del optimizador.

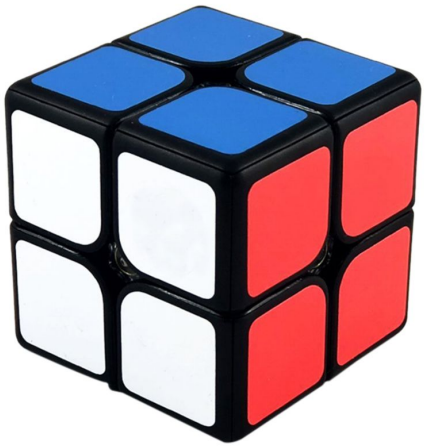
$t \leftarrow 0$;
$InicializarPoblación(t)$;
$EvaluarPoblación(t)$;
**while** *Criterio de parada* **do**
  $t \leftarrow t + 1$;
  $SeleccionarPoblación' desde Población(t-1)$;
  $RecombinarPoblación'$;
  $MutarPoblación'$;
  **if** *Generación en la que aplicar BL* **then**
    $BUSQUEDALOCAL(Población')$;
  **end**
  $ReemplazarPoblación(t) a partir de Población(t-1) y Población'$;
  $EvaluarPoblación(t)$;
**end**

**Algorithm 4:** Algoritmo Memético

El pseudocódigo anterior podría ser una implementación simple de un algoritmo memético. Aunque se trata de un concepto muy general como para que haya una fórmula en la que hibridar estos dos algoritmos.

# Capítulo 5

# Cubo de Rubik 2x2x2



## 5.1. Introducción

El Cubo de Rubik 2x2x2 es una reducción del famoso Cubo de Rubik (3x3x3) inventado por Erno Rubik. Dicho cubo consta de 6 colores (habitualmente: blanco, amarillo, rojo, naranja, verde y azul) y en su posición resuelta cada cara es de un color.

Su mecanismo consiste en 8 piezas que llamaremos «esquinas» que se van moviendo sobre un eje que esta en el centro del cubo. Cada pieza contiene 3 pegatinas, por ejemplo: una pieza/esquina sería la de colores blanco, azul y rojo, que tenemos justo enfrente en la imagen de arriba.

El número de configuraciones posibles es de 3 674 160. Cualquier permutación de 8 esquinas es posible (8!) y 7 de esas esquinas pueden ser rotadas independientemente ($3^7$) puesto que la orientación de la última dependerá del resto. Pero como no hay nada que caracterice la orientación de este cubo, se reducen las posiciones por un factor de 24 (cada una de las 6 caras puede ser base sobre la que se apoya en el suelo, y para cada una de esas 6 posibilidades, podría presentarnos 4 rotaciones en sobre el eje Y). Luego:

$$\frac{8! \times 3^7}{24} = 3\ 674\ 160$$

configuraciones posibles.

## 5.2. Movimientos

Los movimientos posibles para el Cubo de Rubik 2x2x2 son los siguientes:

- **U (Up)**: mover la capa de arriba 90º en setido horario.
- **D (Down)**: mover la capa de abajo 90º en setido horario.
- **R (Right)**: mover la capa de la derecha 90º en setido horario.
- **L (Left)**: mover la capa de la izquierda 90º en setido horario.
- **F (Front)**: mover la capa de enfrente 90º en setido horario.
- **B (Back)**: mover la capa de atras 90º en setido horario.

Con estos movimientos podriamos resolver el 2x2x2. Pero si queremos girar la capa de arriba 90º en sentido antihorario tendríamos que hacer una sucesión de movimientos: **U U U**. Esto, en terminos de ejecución, nos hacen perder mucho tiempo. Por lo tanto hay que definir giros de capas con todos los grados posibles.

Aunque también hay que comentar un detalle más: girar la capa de arriba en sentido horario es lo mismo que girar la capa de abajo en sentido horario. Entonces: **U = D**, por lo tanto, podemos suprimir los giros opuestos (U = D, R = L y F = B). Esto disminuirá el número de posibilidades de movimientos dada una posición del cubo. Con lo cual, estamos reduciendo el tiempo de exploración en computación. Asi que, por proximidad de observación voy a seleccionar los movimientos **U, R y F**. Ahora, mostraremos los movimientos definitivos para el 2x2x2:

- **U**: mover la capa de arriba 90º en setido horario.
- **U'**: mover la capa de arriba 90º en setido antihorario.
- **U2**: mover la capa de arriba 180º.

- **R**: mover la capa de la derecha 90º en setido horario.
- **R'**: mover la capa de la derecha 90º en setido antihorario.
- **R2**: mover la capa de la derecha 180º.
- **F**: mover la capa de enfrente 90º en setido horario.
- **F'**: mover la capa de enfrente 90º en setido antihorario.
- **F2**: mover la capa de enfrente 180º.

Se ha calculado que con esta nomenclatura de movimientos se puede resolver el 2x2x2 desde cualquier configuración en un máximo de 11 movimientos. A esto se le llama el número de Dios del 2x2x2, que es igual a 11.

## 5.3. Busqueda en Anchura

Para representar el 2x2 he usado dos vectores, uno de permutación y otro de orientación. Con el cubo resuelto los vectores estarían tal que así:

- perm = (0, 1, 2, 3, 4, 5, 6, 7)
- ori = (Y, Y, Y, Y, Y, Y, Y, Y)

La esquina UBL es la 0, entendiendo por UBL la esquina que esta en la intersección de las tres capas U, B y L. La 1 es la UBR, la 2 es la UFR, la 3 es la UFL, la 4 es la DFL, la 5 es la DFR, la 6 es la DBR y la 7 es la DBL. Luego para la orientación tendríamos en que eje está la pegatina amarilla o blanca de la pieza en cuestión.

Partiendo de un nodo raíz que contendrá el 2x2x2 mezclado, hay que generar todos los hijos para ese nodo que serán 9, uno por cada movimiento posible. Y estos hijos contendrán la posición del cubo tras hacer el movimiento de transición. Estos hijos a su vez tendrán 6 hijos, puesto que no es posible que dos movimientos consecutivos actúen sobre la misma cara. Como vemos, la observación que hemos hecho antes de reducir los movimientos quitando los opuestos y eliminando la adyacencia de movimientos de capas iguales, nos está ahorrando generar un gran número de nodos. Y en la Búsqueda en Anchura es un aspecto crucial para que no se llene la

memoria por una cantidad enorme de nodos en la cola.

```
raiz ← MEZCLAR(Cubo);
Cola.INSERT(raiz);
while !RESUELTO(inicial) do
    inicial ← Cola.FRONT();
    for M como MovimientosPosibles do
        actual ← inicial.MOVER(M);
        Cola.PUSH(actual);
    end
end
```

**Algorithm 5:** Búsqueda en Anchura

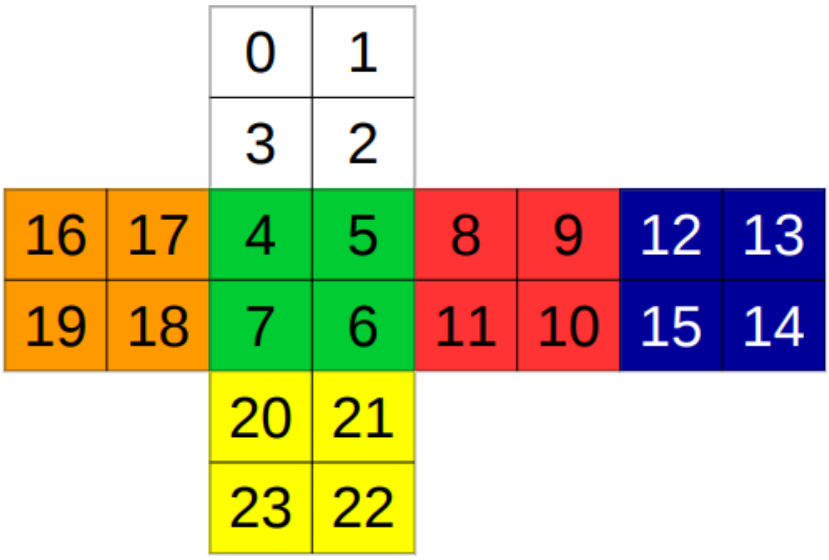
La búsqueda en anchura era solo para tener una referencia del tiempo de ejecución. Ahora empiezo con el estudio del problema desde el planteamiento de las metaheurísticas.

## 5.4. Metaheurísticas

### 5.4.1. Representación del 2x2x2 y de la solución

Comenzamos con el abordamiento del problema usando algoritmos metaheurísticos. Para representar un estado del cubo he usado un vector de 24 caracteres que representan los colores. El caracter **W** representa el color blanco (White), el caracter **Y** representa el color amarillo (Yellow), el caracter **G** representa el color verde (Green), el caracter **B** representa el color azul (Blue), el caracter **R** representa el color rojo (Red), el caracter **O** representa el color naranja (Orange).

Y las posiciones del vector en el cubo se disponen de la siguiente forma:



Luego si el vector está así: **(W, W, W, W, G, G, G, G, R, R, R, R, B, B, B, B, O, O, O, O, Y, Y, Y, Y)** significa que el cubo está resuelto.

Las soluciones van a ser un vector de movimientos de un determinado tamaño. Estos movimientos serán ejecutados en orden sobre la estructura del cubo para ir alterándolo. Un ejemplo de solución sería: (U, F2, R', F', U2, R).

### 5.4.2. Heurística y Función objetivo

El cambio de representación ha sido para favorecer a la heurística. Si tuviese la permutación y orientación separadas, podría tener todas las esquinas en su lugar pero desorientadas y viceversa. Esto nos dificulta encontrar una heurística acorde a la calidad de la solución. Y como ya dijimos en su sección, una función heurística debe ser rápida de ejecutar.

Pero la búsqueda de una heurística que le pueda servir a las Metaheurísticas es una tarea ardua. Puesto que, si estamos a un solo movimiento de la solución va a haber 8 de las 24 pegatinas mal. Y estando a dos movimientos de la solución podemos tener 18 pegatinas de 24 que no tienen el color de su cara. Aquí es donde nos encontramos con el mayor problema a la hora de afrontar el Cubo de Rubik con metaheurísticas. Porque para encontrar una buena heurística habría que hacer un estudio considerable sobre el problema. Y después de dicho esfuerzo quizá una heurística simple funcione mejor, porque hay que tener muy en cuenta la eficiencia.

Otra forma que se me ocurrió, fue la de ver cuántos colores distintos hay en cada cara y ponderar cada cantidad para despues sumarlas. Es decir, con un movimiento que hagamos tenemos las siguientes cantidades de colores en las caras: (1,2,2,2,2,1). Luego, esto puede parecer mejor a simple vista, pero sigue estando lejos del (1,1,1,1,1,1) que sería el estado resuelto. Y con dos movimientos podriamos tener el siguiente vector: (3,3,2,3,2,3), que podría ser igual que el de cualquier mezcla. También pensé en tener en cuenta bloques de piezas en los que las dos pegatinas de cada esquina que se unen sean del mismo color. Pero esto es costoso de comprobar, y tampoco se obtienen resultados muy buenos. La suma de las distancias que hay desde una esquina a su lugar es un caso parecido. No se obtienen buenos resultados y es laborioso de calcular.

Pero esto anterior fue una observación que hice sin tener en cuenta las mutaciones que se llevan a cabo en las metaheurísticas. Cuando estás haciendo una búsqueda IDA* vas construyendo la solución poco a poco, y así nos serviría este enfoque. Pero en nuestro caso, partimos de una solución que tiene una cantidad determinada de movimientos. Y al aplicar esa solución nos deja el 2x2x2 en una configuración concreta. Nosotros vamos a tener que hacer variaciones en cualquier parte de la solución para movernos por el entorno en el espacio de soluciones. Entonces, puede ser que haciendo un cambio en el movimiento $I$ siendo $I$ menor que el número de movimientos

$N$, se nos resuelva el cubo. Este planteamiento anula la explicación de toda heurística comentada en el parrafo anterior. Por lo tanto, me decidí por la simpleza y rapidez de comprobación de la cantidad de pegatinas que están bien.

### 5.4.3. Generación de vecinos

Para la generación de vecinos debemos intentar que las soluciones vecinas tengan un valor heurístico parecido a la solución origen. Esto nos garantizará pocos escalones en los valores del espacio de soluciones y una fácil escalada o caída hacia soluciones.

Encontré dos maneras de disminuir los posibles desfases entre los valores heurísticos cercanos:

- Cambiar un movimiento de la solución por otro que actúe sobre la misma capa. Esto es, tendríamos 3 conjuntos de movimientos: $A_1 = (U, U', U2)$ $A_2 = (R, R', R2)$ y $A_3 = (F, F', F2)$. Solo podremos cambiar un movimiento por otro de su mismo conjunto.

- Intercambiar movimientos adyacentes de lugar. Es decir, colocar el movimiento $i + 1$ en $i$ e $i$ en $i + 1$.

De esta forma, la variación de pegatinas va a ser como máximo de 12 pegatinas. Aunque en su mayoría será de entre 7 y 10, comprobado empíricamente. Antes hemos visto que un movimiento con el cubo resuelto colocaba 8 pegatinas mal, así que, estas dos formas no se pueden considerar malas. Como ya comenté, no es tarea simple amoldar este problema para su tratamiento con metaheurísticas.

### 5.4.4. Búsqueda Local Multiarranque

La Búsqueda en Anchura funcionaba, pero realmente este proyecto se centra en el uso de Metaheurísticas para resolver el Cubo de Rubik. Así que, decidí empezar por una fácil y rápida de implementar. Además de tener en cuenta su fácil ajuste de equilibrio entre exploración y explotación.

Una característica de este problema es que contamos con muchas soluciones en el espacio de soluciones que en realidad no resuelven el cubo. Teniendo en cuenta el número de configuraciones posibles del 2x2x2 nos podemos hacer una idea. Mientras que normalmente, el uso de Metaheurísticas se usa en la optimización de soluciones. En el ejemplo del TSP, cualquier permutación de ciudades era solución, pero había que optimizar la distancia.

En el problema del Cubo de Rubik optimizamos el número de pegatinas que están en su cara. Pero no nos vale solo con optimizar esto, tenemos que hallar al menos una secuencia que tenga las 24 pegatinas en su cara correspondiente. Por lo tanto, para no confundirnos vamos a diferenciar

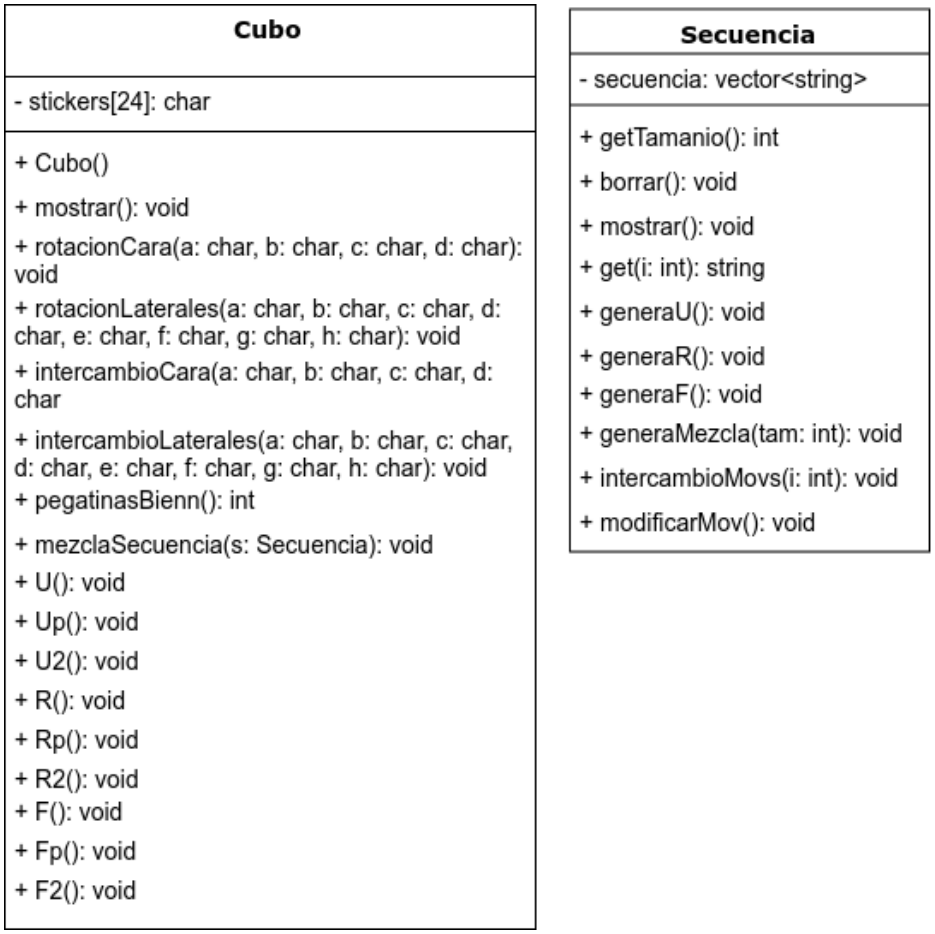
entre **secuencias** y **soluciones**. Todas las soluciones son secuencias, pero no todas las secuencias son soluciones. El espacio de «soluciones» está formado por secuencias. Las secuencias que resuelvan el 2x2x2 serán soluciones y las que no las llamaremos **mezclas**.

$$|SECUENCIAS| = |SOLUCIONES| + |MEZCLAS|$$

Entonces, en un cubo mezclado al azar lo normal es que haya entre 5 y 8 pegatinas bien. La esquina DLB no se mueve cuando lo mezclamos (porque solo usamos movimientos de las capas R, U y F), con lo cual, esas 3 pegatinas siempre van a estar bien. Y como hemos dicho antes, un vecino estándar de una solución tendrá entre 14 y 17 pegatinas bien. Entonces nos encontramos en un espacio de soluciones formado por secuencias que en su gran mayoría tendrán un valor heurístico entre 5 y 17, estas por tanto serán mezclas. Por otro lado, tendremos a las soluciones que serán una inmensa minoría y que tendrán un valor heurístico de 24. No parece un espacio de soluciones ideal para hacer una busqueda mediante alguna metaheurística. Lo normal en los problemas a los que se aplican metaheurísticas es conformarnos con un óptimo local que sea bastante cercano al global. Esto es, nos vale una solución cercana al óptimo teniendo en cuenta la complejidad del problema. En nuestro caso, para 2x2x2 no es un espacio de soluciones grande y podemos encontrar el óptimo global con cierta facilidad. Pero como ya veremos, para el 3x3x3 no será tarea fácil y habrá que optar por otras metodologías.

Como ya sabemos, es bueno mantener un equilibrio entre exploración y explotación. Pero en este caso, vemos bastante claro que hay que decantarse por una mayor exploración. La explotación debe ser de corta duración, puesto que la escalada hacía un óptimo local no es nada complicado desde cualquier punto. Pero nosotros queremos encontrar un óptimo global y solo nos sirve esto. Hay que centrarse en la exploración he intentar llegar rápido al mayor número de óptimos locales diferentes. Llegará el óptimo local que sea también óptimo global y podremos finalizar nuestro seguimiento hacia una solución.

## 5.5. Diagrama de clases




### 5.5.1. Secuencia

La clase **Secuencia** se encarga de todo lo que tiene ver con la ya explicada secuencia. El espacio de soluciones esta formado por instancias de esta clase.

- **secuencia**: se trata de un vector de la clase vector de la STL. Este vector esta compuesto por Strings que serán movimientos. Rp equivale a R', Fp a F' y Up a U'.
- **getTamanio()**: devuelve el tamaño del vector de movimientos.
- **borrar()**: borra el contenido de la secuencia.
- **mostrar()**: muestra por la terminal la secuencia.
- **get(i)**: devuelve el string posicionado en el índice «i» de la secuencia.

- **generaU()**: añade a la secuencia uno de los 3 movimientos relativos a la capa U.
- **generaF()**: añade a la secuencia uno de los 3 movimientos relativos a la capa F.
- **generaR()**: añade a la secuencia uno de los 3 movimientos relativos a la capa R.
- **generaMezcla(tam)**: genera una secuencia de «tam» movimientos.
- **intercambioMovs(i)**: intercambia los movimientos $i$ e $i + 1$ de la secuencia.
- **modificarMov()**: modifica un movimiento aleatorio de la secuencia. Solo modifica los grados del giro de dicho movimiento.

### 5.5.2. Cubo

Mientras que la clase **Cubo** maneja todo lo relativo al cubo y su movimiento.

- **stickers**: vector estático de char que contiene a los stickers.
- **pegatinasBien()**: devuelve un entero con el número de pegatinas que están en la cara correcta.
- **rotacionCara(a,b,c,d)**: permuta los parámetros de forma que se produzca un desplazamiento de 1. Se aplica para permutar las 4 pegatinas de la parte superior de la capa
- **rotacionLaterales(a,b,c,d,e,f,g,h)**: vuelve a permutar los parámetros, pero esta vez tiene en cuenta que son pegatinas laterales de la capa.
- **intercambioCara(a,b,c,d)**: este método se usa para giros dobles, con lo cual intercambia a-c y b-d.
- **intercambioLaterales(a,b,c,d,e,f,g,h)**: teniendo en cuenta que es un giro doble intercambia las pegatinas laterales de la capa.
- **métodos relativos a los movimientos**: realizan los movimientos sobre el vector stickers. Son 9 funciones: U(), Up(), U2(), R(), Rp(), R2(), F(), Fp() y F2().
- **mezclaSecuencia(s)**: mezcla el cubo siguiendo los movimientos de la secuencia «s».

# Capítulo 6

# Cubo de Rubik 3x3x3

## 6.1. Introducción

En este capítulo vamos a trabajar sobre el famoso Cubo de Rubik de 3x3x3. Se suele conformar de los mismos colores que el 2x2x2: amarillo, blanco, naranja, rojo, azul y verde. En el 2x2x2 solo teniamos un tipo de pieza, la **esquina**. Para este rompecabezas contamos con 2 tipos de piezas más. La primera que voy a comentar y más imporante es el **centro**. Hay un solo centro en cada cara, con lo cual, hay seis centros. El color de cada centro es el que dictamina que color debe ir en esa cara puesto que los centros siempre están colocados en el mismo lugar con respecto otro centro. Un centro está formado por la pegatina que está en el centro de cada cara. Por otro lado, tenemos las **aristas**, que son las piezas que estan entre las esquinas. Estas piezas están formadas por 2 pegatinas y hay un total de 12

aristas en el 3x3x3.

- **Aristas**: UB, UR, UF, UL, BL, BR, FR, FL, DB, DR, DF y DL.
- **Esquinas**: UBL, UBR, UFR, UFL, DBL, DBR, DFR y DFL.

Para el 2x2x2 teniamos una cantidad de 3 674 160 configuraciones posibles. Para este puzzle hay 8! formas de combinar las esquinas del cubo. 7 de estas pueden orientarse individualmente, la orientación de la octava dependerá del resto ($3^7$). Por otra parte, hay 12!/2 maneras de organizar las aristas dado que una paridad en las aristas implica una paridad en las esquinas. Y con la orientación de las aristas sucede lo mismo que con la de las esquinas ($2^{11}$):

$$\frac{8! \times 3^7 \times 12! \times 2^{11}}{2} = 43\ 252\ 003\ 274\ 489\ 856\ 000$$

configuraciones posibles.

## 6.2. Movimientos

Los movimientos necesarios para llegar a todas las configuraciones del Cubo de Rubik 3x3x3 son los siguientes:

- **U (Up)**: mover la capa de arriba 90º en setido horario.
- **D (Down)**: mover la capa de abajo 90º en setido horario.
- **R (Right)**: mover la capa de la derecha 90º en setido horario.
- **L (Left)**: mover la capa de la izquierda 90º en setido horario.
- **F (Front)**: mover la capa de enfrente 90º en setido horario.
- **B (Back)**: mover la capa de atras 90º en setido horario.

No vamos a tener en cuenta los movimientos de las capas intermedias. Como hemos dicho antes, los centros no se mueven respecto de los otros, así que, es una oportunidad para eliminar movimientos posibles. De esta forma, limitamos el conjunto de movimientos, que va a ser una tarea pendiente a la hora de resolver el puzzle usando Metaheurísticas. Aunque, como necesitamos ahorrar tiempo de ejecución como en el 2x2x2, tendremos un total de 18 movimientos:

- **U**: mover la capa de arriba 90º en setido horario.
- **U'**: mover la capa de arriba 90º en setido antihorario.
- **U2**: mover la capa de arriba 180º.

- **R**: mover la capa de la derecha 90º en setido horario.
- **R'**: mover la capa de la derecha 90º en setido antihorario.
- **R2**: mover la capa de la derecha 180º.
- **F**: mover la capa de enfrente 90º en setido horario.
- **F'**: mover la capa de enfrente 90º en setido antihorario.
- **F2**: mover la capa de enfrente 180º.
- **L**: mover la capa de la izquierda 90º en setido horario.
- **L'**: mover la capa de la izquierda 90º en setido antihorario.
- **L2**: mover la capa de arriba 180º.
- **B**: mover la capa de atrás 90º en setido horario.
- **B'**: mover la capa de atrás 90º en setido antihorario.
- **B2**: mover la capa de atrás 180º.
- **D**: mover la capa de abajo 90º en setido horario.
- **D'**: mover la capa de abajo 90º en setido antihorario.
- **D2**: mover la capa de abajo 180º.

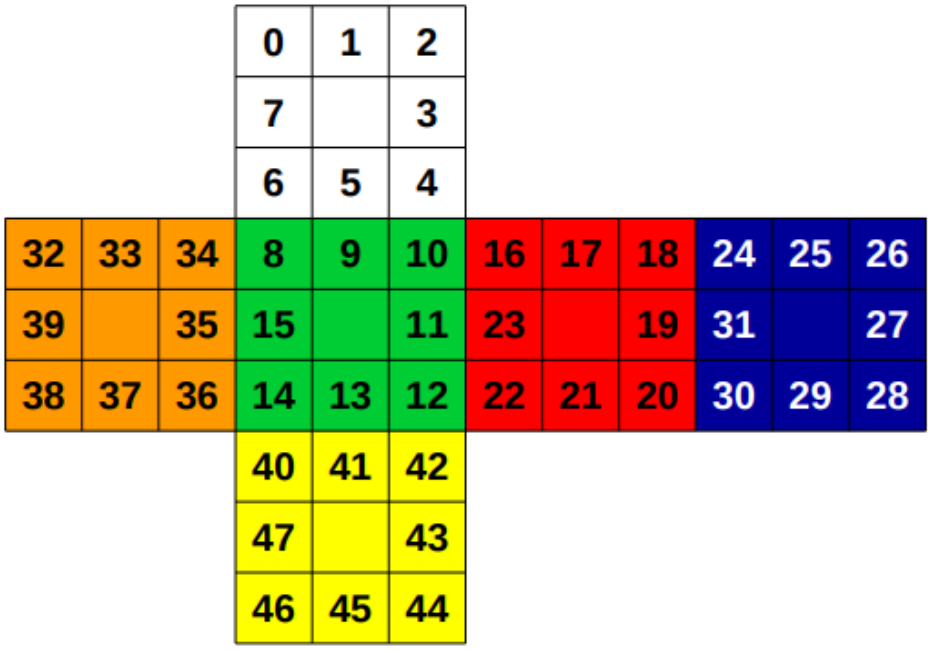
Según estudios probados mediante la informática se ha calculado que el número de Dios del 3x3x3 es 20 con esta lista de movimientos. De esta forma, podemos resolver cualquier configuración del 3x3x3 usando como máximo 20 movimientos. Esto se calculó en 2010 y el tiempo de computación fue alrededor de 3 semanas, para probar todas las configuraciones.

## 6.3. Metaheurísticas

### 6.3.1. Representación del 3x3x3 y de la solución

Para representar el 3x3x3 he usado el mismo sistema que empleé para el 2x2x2. Y tiene la misma explicación, solo que ahora se fundamenta en los trabajos que usan Algoritmos Genéticos para resolver este puzzle, y que utilizan esta misma representación. Y es que, como comenté anteriormente, la simplicidad en la representación es muy importante.

Y los índices del vector en el cubo se distribuyen de la siguiente manera:




De esta forma, el vector con el 3x3x3 resuelto será: **(W, W, W, W, W, W, W, W, G, G, G, G, G, G, G, G, R, R, R, R, R, R, R, R, B, B, B, B, B, B, B, B, O, O, O, O, O, O, O, O, Y, Y, Y, Y, Y, Y, Y, Y)**.

### 6.3.2. Heurística y función objetivo

Antes de empezar a intentar resolver este problema, se me ocurrió una idea para encontrar una heurística de mayor calidad. La idea consistía en usar Machine Learning para ajustar los parámetros de ciertas características del Cubo de Rubik. Empezaría generando todas las secuencias de hasta $N$ movimientos que me permitiese una búsqueda en profundidad. Puesto que la búsqueda en anchura para el 3x3x3 llena mi memoria cuando está explorando secuencias de 6 movimientos. Con una búsqueda en profundidad he logrado encontrar secuencias de hasta 7 movimientos en poco tiempo, y creo que con una hora o dos más podria encontrar secuencias de 8 movimientos. Entonces, teniendo una serie de características del puzzle como pueden ser: número de pegatinas en su cara, número de esquinas en su sitio, número de aristas orientadas, cantidad de colores en la cara F,... Podría hallar más de 50 características relativas al cubo y su configuración. Y el número de movimientos para llegar a esa configuración sería la variable a predecir. Con todo esto, podemos aplicar técnicas de aprendizaje automático para predecir el número de movimientos hasta la solución teniendo las características necesarias. Pero había varios problemas para intentar esto:

- No tenía certeza de que pudiese predecir con un acierto alto el número de movimientos. De hecho, viendo la complejidad del problema diría que el porcentaje de acierto sería bastante bajo.
- El tiempo en calcular las distintas características podría ser más de 30 veces superior al de la heurística de contar cuantas pegatinas están en la cara de su color.
- Para posiciones más alejadas de 8 movimientos no servía el estudio.

En consecuencia, usé la heurística que han utilizado la mayoría de los trabajos que he inspeccionado. El número de pegatinas que tienen el color igual al de su cara. Como ya dije anteriormente, es un factor clave que una heurística sea rápida de calcular. Además, en el 3x3x3 una solución y sus vecinas se diferencia en un rango de entre 13 y 16 en el valor heurístico. En el 2x2 el número de pegatinas bien entre una solución y su vecina era mayor en proporción.

### 6.3.3. Generación de vecinos

Para este apartado utilizo lo mismo que para el 2x2x2. En el 2x2x2 un cambio de los que comenté permutaba 4 piezas de sitio. En el 3x3x3 una mutación en la secuencia de las que ya expliqué permutaría 8 piezas. 8 piezas es lo mínimo que podemos permutar en este rompecabezas actuando sobre la secuencia. Luego obtenemos estas dos formas:

- Cambiar un movimiento de la solución por otro que actue sobre la misma capa. Esto es, tendríamos 6 conjuntos de movimientos: $A_1 = (U, U', U2)$, $A_2 = (R, R', R2)$, $A_3 = (F, F', F2)$, $A_4 = (L, L', L2)$, $A_5 = (B, B', B2)$ y $A_6 = (D, D', D2)$. Solo podremos cambiar un movimiento por otro de su mismo conjunto.
- Intercambiar movimientos adyacentes de lugar. Es decir, colocar el movimiento $i + 1$ en $i$ e $i$ en $i + 1$.

De esta forma, la variación de pegatinas va a ser como máximo de 20 pegatinas ante cualquiera de los dos procedimientos. Aunque en su mayoría será de entre 13 y 16.

### 6.3.4. Búsquedad Local Multiarranque

Para el 3x3x3 si que podemos apreciar con mayor facilidad los escalones entre soluciones y sus vecinos. Ya que aquí tenemos un espacio de soluciones enorme en comparación con el del 2x2x2. Una secuencia con valor heurístico de 48 se convierte en una solución, ya que habría 48 pegatinas en la cara correspondiente. Y los vecinos de esta solución van a tener de valor heurístico entre 32 y 35. Mientras que un cubo mezclado con 25 movimientos aleatorios

suele tener entre 4 y 9 de valor heurístico. Por lo tanto, tendríamos una escalada bastante fácil, que sería desde una mezcla absoluta hasta tener cerca de 30 pegatinas en su sitio. Pero luego, tendríamos una cantidad inmensa de secuencias con valor heurístico mayor de 30 y un porcentaje mínimo de que una de ellas esté cerca de una solución. Entonces, conviene centrarse en la exploración ya que la explotación puede ser muy corta y satisfactoria en el caso de estar cerca de una solución.

Aunque como veremos más adelante, no me ha quedado más remedio que fragmentar en etapas el Cubo de Rubik. Si nos hacemos una idea del espacio de soluciones que puede generar el 3x3x3 nos daremos cuenta de la complejidad de este problema.

### 6.3.5. Algoritmo Genético

Para la Búsqueda Local Multiarranque no conseguí resolver el puzzle de una pasada. Pero los Algoritmos Genéticos funcionan de una manera muy dispar. Tendríamos una población de individuos, en nuestro caso secuencias. Cada secuencia está unida a su valor heurístico y ordenada en la población según este valor. Para ello uso una estructura llamada **multimap** que tiene dos componentes. La primera es un float o un int y la segunda es un objeto. Y al insertar un objeto multimap en la población la introduce manteniendo el orden de la primera componente. La primera componente contendrá el valor heurístico. Por lo tanto, vamos a tener siempre ordenada la población según su valor heurístico. Mientras que el segundo elemento incluirá la secuencia de movimientos.

Ahora hay que ir cruzando padres para generar hijos e incluirlos en la generación siguiente. Aquí hay varias opciones:

- Los hijos sustituyen a los padres.
- Los hijos se unen con los padres y seleccionamos a la porción de la población de mejor valor heurístico.
- Al tener los 2 padre y los 2 hijos seleccionamos los dos individuos de mejor valor heurístico.
- El mejor hijo sustituye al peor padre.
- Se generan 3 hijos de dos padres y nos quedamos con los dos mejores.

De estas alternativas he probado las 3 primeras, dado que las dos últimas me parecían similares a alguna de las 3 de arriba. En la primera opción no obtuve buenos resultados comparados con las otras dos. La segunda alternativa mejoró a la primera con creces, puesto que proporcionaba bastante elitismo. Y la tercera es la que mejores resultados me proporcionó. Quizás, por no seleccionar a las mejores secuencias de toda la población. De esta

forma, al seleccionar 2 de un grupo de 4 podemos eliminar cierto grado de elitismo que es importante a la hora de mantener variedad en la población.

Para las mutaciones, también había muchos caminos a seguir:

- Hacer una mutación de generación de vecino.
- Hacer un desplazamiento aleatorio en la secuencia insertando valores del final en el principio.
- Cambiar cada movimiento por su opuesto (me refiero a la cara opuesta).

Y muchas más combinando estas propuestas, como por ejemplo, la de hacer una mutación de generación de vecino que no funcionaba. Esta mutación no genera variedad en la población. De hecho, para que al menos llegase a alguna solución tuve que hacer cada cierto número de generaciones una reiniciación de la población con individuos totalmente nuevos. Opté por ampliar el grado de la mutación y probé la tercera alternativa. Esta alternativa tenía el problema de que al volver a hacer esta mutación resultaba la misma secuencia que tenías. Con lo cual volvía a converger pronto. La segunda opción sí que mejoró bastante los resultados, pero cuando la población iba convergiendo los movimientos eran muy parecidos en todos los individuos. Al final, me decanté por una mutación aleatoria. Es decir, una reinicialización de un porcentaje de individuos de la población. Y esta mutación es la que mejor resultados me daba con diferencia, puesto que no convergía nunca.

Por último, he tenido en cuenta el elitismo de los mejores cromosomas. Y he mantenido entre generaciones las mejores, esto funcionaba bien, puesto que al cruzar con otros padres, podías obtener secuencias similares y de gran calidad. Aunque tras todo lo comentado, tampoco he conseguido resolver el 3x3x3 sin dividirlo en pasos.

### 6.3.6. Algoritmos Meméticos

A veces, puede haber algún individuo de nuestra población que esté cercano a una solución. En estos casos, sería conveniente llevarlo hacia ella o aproximarnos lo máximo posible. Para ello se inventaron los algoritmos meméticos y por eso he decidido aprovecharlos para este problema. Como ya tenía implementada la Búsqueda Local, este ha sido mi algoritmo de explotación.

Empecé haciendo explotación en las mejores secuencias de la población, puesto que pensé que las otras no tenían posibilidades de llegar a la solución. Pero el objetivo es ir mejorando la población poco a poco manteniendo cierta variedad. Y de esta forma, mejoraba solo a un tanto por ciento muy pequeño. Así que decidí aplicarlo a un tanto por ciento de la población, de manera aleatoria. Pero lo que mejor resultados me dio fue emplearlo con la población

completa. De esta forma, no se me escapaba la posibilidad de que un solo individuo estuviese cerca de la solución.

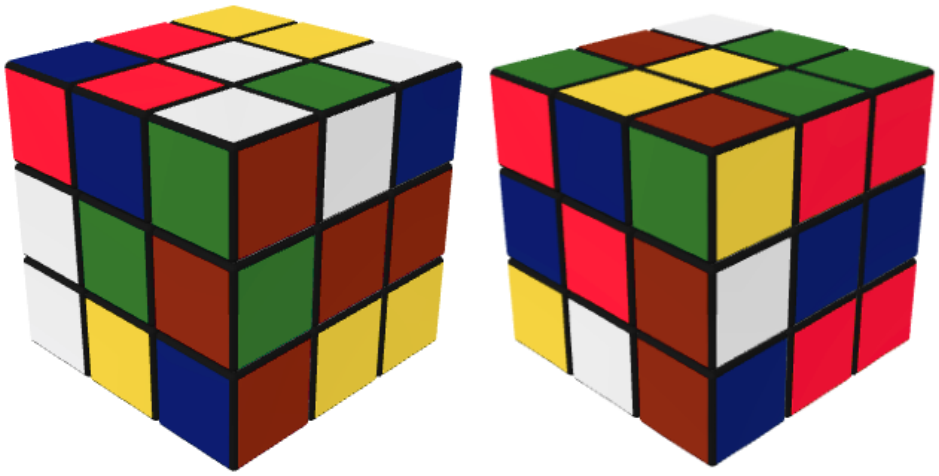
Otro factor a tener en cuenta era la distancia de generaciones hasta el uso de la Búsqueda Local. Al principio, probé con una distancia de 100 generaciones. Y a medida que iba disminuyendo este número el algoritmo funcionaba mejor. Así que, acabé aplicando la Búsqueda Local cada muy pocas generaciones.

Los resultados mejoraron bastante respecto de los Algoritmos Genéticos puros. Mejoró tanto el tiempo de encontrar la solución como el número de movimientos. Aunque, seguía sin poder resolver el 3x3x3 completo.

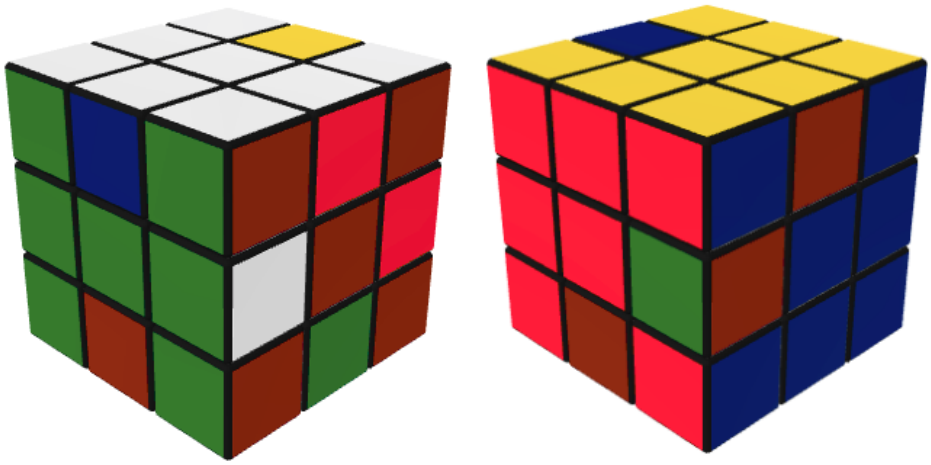
## 6.4. Estudio del problema

### 6.4.1. Resolución en una etapa

Un solo trabajo de los que supervise lograba resolver el Cubo de Rubik en un solo paso usando Metaheurísticas. Y este parecía el reto a conseguir ,o como poco, el primero a probar. Lo implementé y lo dejé ejecutar una noche entera. No consiguió encontrar solución y además se quedó muy lejos de ella. Aquí fue cuando empecé a pensar en la forma de dividirlo en fases. Pero cuando optimicé la resolución en fases ya había conseguido unos parámetros mejores que cuando hice el primer intento. Así que, decidi probarlo:



Este era el cubo mezclado, y como podemos ver, tenemos 10 pegatinas en su cara (sin contar los centros). Y así es como quedó después de 509 minutos ejecutándose el Algoritmo Memético que mejor resultados me proporcionaba:



Quedan 12 pegatinas que no están en su cara y, por el contrario, 36 que están bien. La primera vez que lo intenté sin ajustar los parámetros se quedaron 34 pegatinas con el color correspondiente a su cara. Tras esto, vi que este no era el camino, y que había que dividir el Cubo de Rubik en etapas.

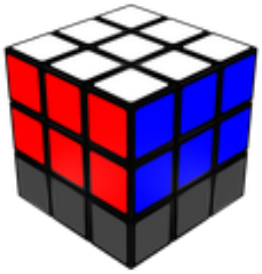
### 6.4.2. Método CFOP

Acto seguido se me ocurrió una forma de particionar el rompecabezas en fases. La manera fue aplicando el conocido, por apasionados de esta afición, método Fridrich (CFOP). Este es el método usado por más del 90 % de los mejores competidores del mundo. Se trata de un método que suele usar entre 50 y 60 movimientos y que consta de 4 etapas:

- **Paso 1: CRUZ**

Consiste en completar una cruz en la capa que quieras. Solo hay que posicionar las 4 aristas de la cara en su sitio, y este paso suele llevar unos 7 movimientos. Lo podriamos solventar con metaheurísticas sin mucho problema.

- **Paso 2: F2L (First 2 Layers)**

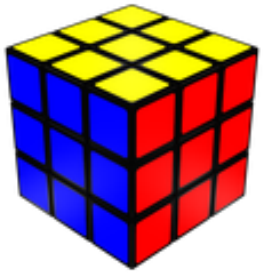


Este fase se compone de 4 subfases en las que vamos a ir colocando «pares de F2L». Un par de F2L es la unión de una esquina con la de una arista que se inserta entre los huecos que vemos en la imagen de la cruz. Cada par nos supone una media de 8 movimientos, pero nunca tendremos que hacer movimientos de la capa D. Aunque tenemos que tener en cuenta que el cubo de la imagen esta volteado para visualizar mejor lo que resuelve esta etapa. La cara blanca deberá ir debajo y vamos a considerar cinco grupos de movimientos: $A_1 = (R, R', R2)$, $A_2 = (F, F', F2)$, $A_3 = (L, L', L2)$, $A_4 = (B, B', B2)$ y $A_5 = (U, U', U2)$. De forma que, para introducir un par de F2L solo necesitaremos un grupo de los 4 primeros y el 5. Además se intercalarán empezando y acabando por uno de los 4 primeros. Por ejemplo, $R\ U\ R'\ U2\ R\ U'\ R'$. Así, no habrá un espacio de soluciones grande y podremos resolver rápido este paso.

- **Paso 3: OLL (Orientation Last Layer)**

En este paso hemos volteado el cubo y colocado la cara blanca abajo. Y consiste en colocar todas las pegatinas amarillas en la cara de arriba. Esto conlleva aproximadamente 10 movimientos de media eliminado movimientos de la capa D. Aunque seleccionando movimientos de la cara U, F y R supongo que se podría solventar en no más de 14 movimientos para cualquier caso.

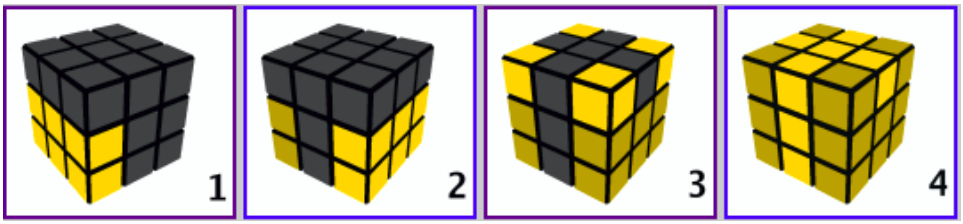
- **Paso 4: PLL (Permutation Last Layer)**



Y este es el paso que nos da bastantes problemas, que consiste en permutar las piezas de la capa de arriba que ya están orientadas. Esta etapa consume una media de 13 movimientos siendo 15 el peor caso. Y sin eliminar ninguna capa de movimiento, con lo cual, nuestra metaheurística no tiene posibilidades de enfrentarse a esto.

### 6.4.3. Método Roux

Vistos los problemas del método anterior, ahora debía encontrar un método que disminuyese el espacio de soluciones y sus etapas no necesitasen muchos movimientos. Y encontré un método acorde a estas caracteristicas llamado **Roux**:




http://grrroux.free.fr/method/Intro.html

- **Paso 1: Primer bloque 1X2X3**

  Construir un bloque 1x2x3 nos lleva unos 9 movimientos de media. Usando todas las capas del Cubo de Rubik y esto es factible porque tan solo debemos tener 12 pegatinas bien y el resto pueden estar de cualquier forma. Aunque se puede dividir en dos etapas si se quiere. bloque de 2x2x1 y después insertar un bloque de 1x2x1.

- **Paso 2: Segundo bloque 1X2X3**

  Para este paso necesitamos aproximadamente unos 13 movimientos, que de nuevo se pueden dividir en subetapas. Aquí se reduce el número de caras a mover, aunque es conveniente usar la capa vertical que está en medio de la cara frontal.

- **Paso 3: COLL (Corner Of Last Layer)**

  Y aquí estamos de nuevo en el paso que nos puede dar mayor problema. Aunque se puede resolver en una media de 10 movimientos usando todas las capas del cubo. Es decir, es viable que el Algoritmo Memético pueda encontrar solución a esta fase.

- **Paso 4: L6E (Last 6 Edges)**

  En esta etapa solo hay que mover la capa central que he dicho antes, que la componen los centros blanco, verde, amarillo y azul junto con las aristas UB, UF, DF y DB. Y también hay que mover la capa de arriba. Pero estas dos capas se intercalarán luego el espacio de soluciones será muy pequeño aunque el número de movimientos para resolver esta fase sean de media 15.

Hacen un total de 47 movimientos de media y vemos viabilidad en todas las capas. Pero justo en este momento, estaba investigando y encontré el método más acorde para dividir el cubo en etapas.

### 6.4.4. Algoritmo Thistlethwaite

El algoritmo Thistlethwaite consiste en ir resolviendo «partes» del cubo para ir descartando movimientos posibles. Este método se basa en la teoría de grupos aplicada a nuestro puzzle.

Al principio, con el cubo mezclado, tendremos todos los movimientos posibles. Esto es:

$$(U, D, R, L, F, B)$$

Que en realidad es lo mismo que tener:

$$(U, U', U2, D, D', D2, R, R', R2, L, L', L2, F, F', F2, B, B', B2)$$

Solo que de la anterior forma ahorramos espacio y queda bien claro. Así que, partimos del grupo $G_0$:

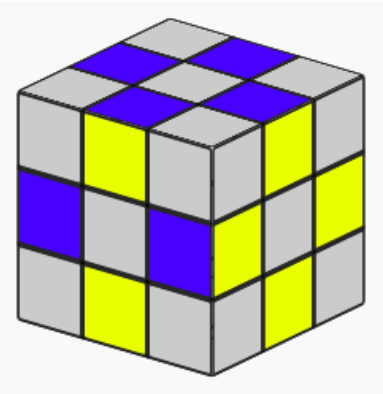
$$G_0 =< L, R, F, B, U, D >$$

Y este grupo contiene todas las posiciones del Cubo de Rubik. Luego este grupo hay que reducirlo, y se hará «orientando» las aristas. Este paso nos llevará de media unos 5 movimientos. No hemos hablado nada de orientación de aristas, pero en este parrafo vamos a dialogar sobre esto. Empezamos con la posición incial del cubo, debe estar el centro blanco en la cara de arriba y el centro verde en la cara de enfrente. Tras esto vamos a diferenciar tres grupos:

- **Colores fuertes**: en nuestro caso van a ser tanto el blanco como el amarillo, por estar en la cara de arriba y de abajo.

- **Colores semifuertes**: estos serán el verde y el azul, que están en la cara frontal y trasera.
- **Colores débiles**: serán el naranja y el rojo, por su posicionamiento en las caras laterales.

Y ahora veamos la siguiente imagen:



Sobretodo, debemos fijarnos en la posición de las pegatinas azules. Por supuesto en la cara de abajo estarán 4 pegatinas azules tal y como en la cara de arriba. Y además en la cara de atras-derecha habrá 2 pegatinas azules tal y como en la cara frontal-izquierda. Es decir, tendríamos un total de 12 pegatinas azules que en realidad son posiciones de pegatinas del Cubo de Rubik. No quiere decir, que nuestro cubo tenga esas 12 pegatinas azules, ya que es imposible. Para no confundirnos, llamaremos a estas pegatinas **Posiciones Orientadas**. Lo importante es ver donde están localizadas las Posiciones Orientadas en nuestro rompecabezas. Pues la norma que tenemos que seguir para tener todas las aristas orientadas es muy simple. En las Posiciones Orientadas debe haber una pegatina fuerte o semifuerte, nunca una débil. Y si hay una pegatina semifuerte, es porque la otra pegatina de la arista no es fuerte. Es decir, habrá estos tipos de aristas según sus pegatinas:

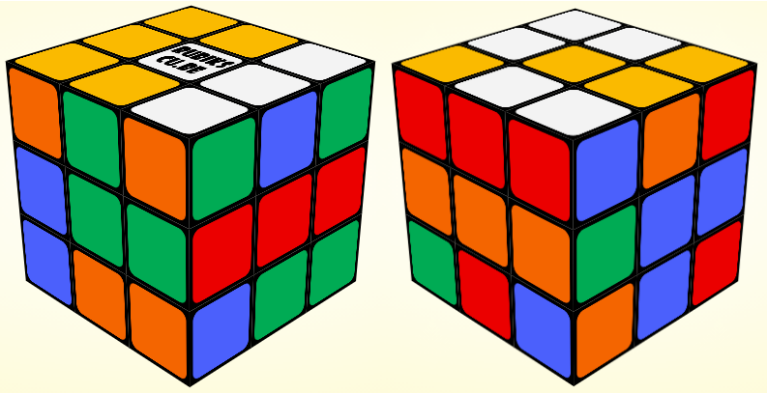
- **Fuertes-Semifuertes**: (Blanco-Verde, Blanco-Azul, Amarillo-Verde, Amarillo-Azul). La pegatina fuerte debe ir en la Posicion Orientada.
- **Fuertes-Debiles**: (Blanco-Rojo, Blanco-Naranja, Amarillo-Rojo, Amarillo-Naranja). La pegatina fuerte debe ir en la Posición Orientada.
- **Semifuertes-Debiles**: (Verde-Rojo, Verde-Naranja, Azul-Rojo, Azul-Naranja). La pegatina semifuerte debe ir en la Posición Orientada.

Orientar las aristas nos reduciría el espacio de soluciones desde las $4,33 \times 10^{19}$, aproximadamente, a las $2,11 \times 10^{16}$ configuraciones posibles. Al orientarlas, reducimos el grupo de movimientos posibles a:

$$G_1 =< L, R, F2, B2, U, D >$$

Es decir, para las capas F y B solo debemos hacer giros dobles. Para el resto, podemos hacer cualquier giro. Teniendo cualquier posición del Cubo de Rubik con las aristas orientadas podemos resolverlo usando los movimientos de $G_1$.

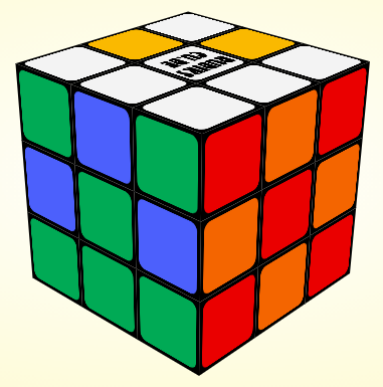
Teniendo las aristas orientadas, el siguiente paso consiste en orientar las esquinas y conseguir que tanto en la cara U y la cara D todas las pegatinas sean o blancas o amarillas:



Esto se debe conseguir mediante giros del grupo $G_1$. Este paso nos costará de media unos 8 movimientos. Cuando consigamos este cometido habremos reducido el espacio de soluciones a $1,95 \times 10^{10}$ configuraciones posibles. Y el grupo de movimientos disminuirá a:

$$G_2 =< L2, R2, F2, B2, U, D >$$

Ahora debemos lograr que en las caras L y R solo haya pegatinas de color rojo o naranja. En las caras U y D solo debe haber pegatinas amarillas o blancas y en las caras F y B que solo contengan pegatinas verdes o azules. Además de esto, deberemos posicionar cada esquina en su lugar. Nos quedaría algo como esto:



De esta forma, restringimos nuestro espacio de soluciones al tamaño de $6,63 \times 10^5$ configuraciones posibles. Este paso nos llevará unos 11 movimientos de media. Y nuestro nuevo grupo de movimientos será:

$$G_3 =< L2, R2, F2, B2, U2, D2 >$$

Y con este grupo de 6 movimientos que representan únicamente a los giros dobles podemos resolver el Cubo de Rubik con una media de 10 movimientos.

### 6.4.5. Algoritmo Kociemba

El Algoritmo Kociemba trata de reducir los pasos del Algoritmo Thistlethwaite a 2. Es decir, agrupar los 2 primeros pasos de Thistlethwaite en 1 solo paso y hacer lo mismo con los 2 siguientes pasos. El primer paso reduciría el grupo de movimientos a:

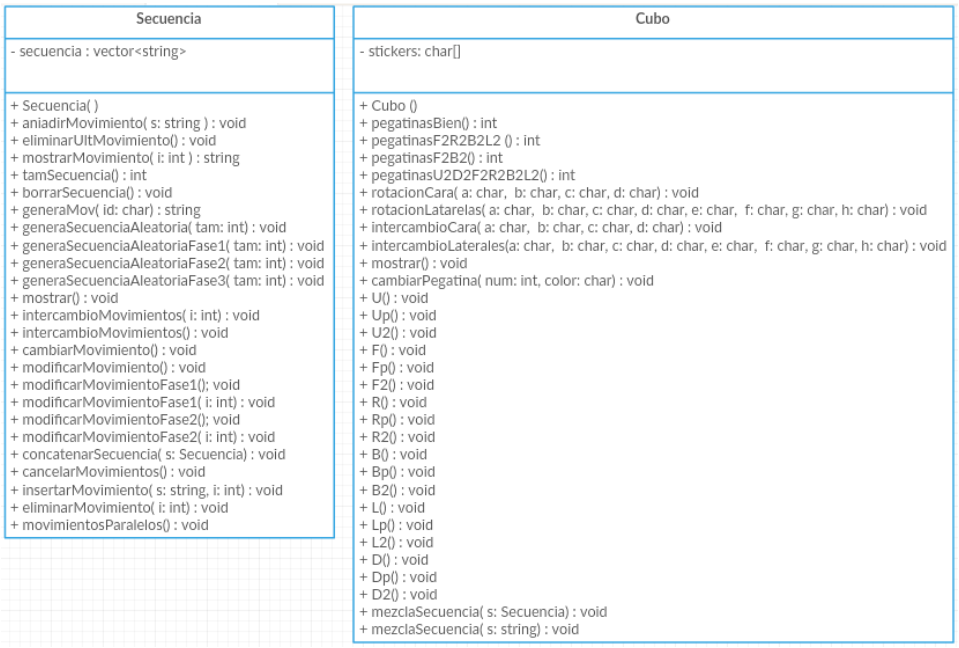
$$G_1 =< L2, R2, F2, B2, U, D >$$

En el que tendríamos que tener todas las pegatinas de las caras U y D tanto blancas como amarillas y el resto de aristaas orientadas, tal y como el segundo paso de Thistlethwaite. Esto nos tomaría una media de 11 movimientos. Y el resto del cubo se resolvería aplicando movimientos de $G_1$. Que nos llevaría alrededor de 18 movimientos.

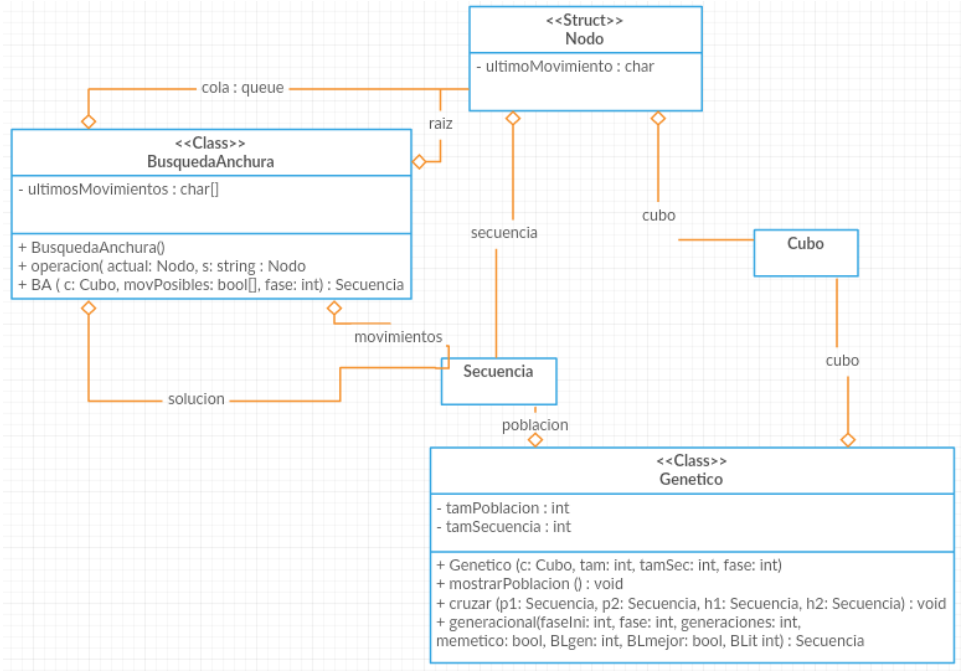
El primer paso con Metaheurísticas es viable y no tarda demasiado. Pero en el segundo paso, sí que se demora mucho debido a la cantidad de movimientos. Debido a esto, se me ocurrió una hibridación entre Kociemba y Thistlethwaite. Hago el primer paso de Kociemba y los dos últimos pasos de Thistlethwaite. Así, no tarda demasiado usando Metaheurísticas y reduzco el número de movimientos totales. Lo de que el número de movimientos totales disminuye está demostrado, pero que mi metaheurística encuentre el óptimo es más complicado. La media de movimientos de los dos primeros pasos del Algoritmo Thistlethwaite son 13 mientras que la media de movimientos del primer paso del Algoritmo Kociemba es 11.

## 6.5. Diagrama de clases

Empezaremos con las dos clases imprescindibles para la construcción de este problema:




Las clases Secuencia y Cubo son bastante parecidas a las clases usadas para el problema del 2x2x2. La clase Cubo tiene varios métodos que calculan el número de pegatinas bien. Cada método hace alusión a un paso distinto del Algoritmo Thistlethwaite. El nombre de las funciones hace referencia a los giros dobles que solo se permiten en la siguiente etapa del algoritmo. Después, tenemos un método nuevo llamado «cambiarPegatina» y su función es cambiar una pegatina perteneciente a su índice. Esto ya veremos para que se usa en el Manual de Uso. La clase Secuencia es también muy parecida a la que ya utilizamos para el 2x2x2. Aunque, al generar secuencias debemos tener en cuenta la fase en la que estamos para no incluir movimientos que ya no son necesarios. Y en la de modificar movimientos sucede lo mismo. Tenemos dos métodos llamados «concatenarSecuencia» y «concatenarMovimientos». Estas dos funciones unirán las distintas secuencias de las etapas y si en la unión se pueden eliminar movimientos por repetición, se suprimirán. «movimientosParalelos» trata de eliminar movimientos en los que se giran capas opuestas 3 veces consecutivas. Por ejemplo, **U2 D U'** es lo mismo que **U D**.

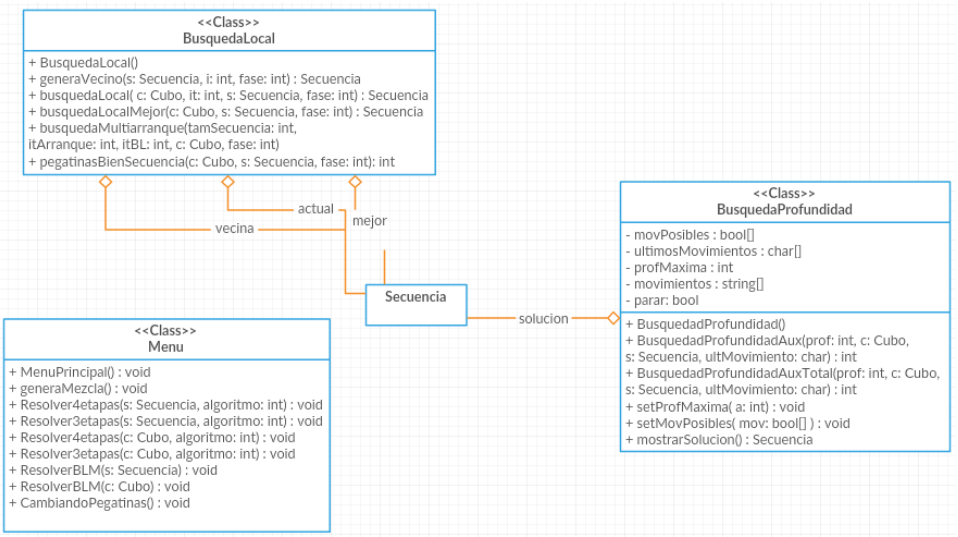



En este diagrama he simplificado los cuadros de las clases ya explicadas por su larga extensión. Comenzaremos hablando de el Struct Nodo, que es la base para hacer la búsqueda en anchura. He de decir que la búsqueda en anchura para el 3x3x3 no la he usado en ninguna parte de este proyecto. Pero quise ver cual era su capacidad, la de llegar a una cantidad de 6 movimientos y que el ordenador se bloqueara. Esto es, a mi parecer, al gran uso de memoria que consumen los nodos en la cola. Cada nodo, tendrá un objeto Cubo, que representará su configuración. Contendrá una secuencia de movimientos hasta llegar a ese nodo. Y el último movimiento realizado sobre el cubo para no repetir movimiento de capa en movimientos consecutivos.

En la clase BúsquedaAnchura tenemos una cola de nodos, de la que iremos metiendo y sacando nodos. Un nodo raiz, del que partimos, una secuencia de los movimientos que llevamos y la secuencia de solución. Luego, incluimos un método de operación que se encarga de las transiciones entre nodos y un método que realiza la búsqueda en anchura. Mientras que en la clase Genetico poseemos un multimap «población», para tener ordenados a los cromosomas según su valor heurístico. También tendremos que tener un objetivo Cubo, el tamaño de la población y el tamaño de la secuencia de movimientos (según la fase en la que estemos). Los métodos son:

- **Genetico()**: constructor que inicia la población.
- **mostrarPoblacion()**: muestra la población con la finalidad de visualizar que todo vaya bien.
- **cruzar()**: Mediante dos padres, genera dos hijos con un cruce partido.

- **generacional()**: realiza la evolución de la población de tipo generacional.




Por último, tenemos la clase BusquedaProfundidad, de la cual no voy a hablar demasiado puesto que solo la hice para ver su capacidad. Un poco mejor que la Búsqueda en Anchura llegando a los 7 movimientos de profundidad en no demasiado tiempo. La clase BusquedaLocal es prácticamente igual que la ya empleada para el 2x2x2. Y la clase Menu tampoco tiene mucho que comentar puesto que se encarga de la funcionalidad del menú. Métodos que imprimen en la terminal menús para ir leyendo opciones seleccionadas por el usuario.

# Capítulo 7

# Manual de Usuario

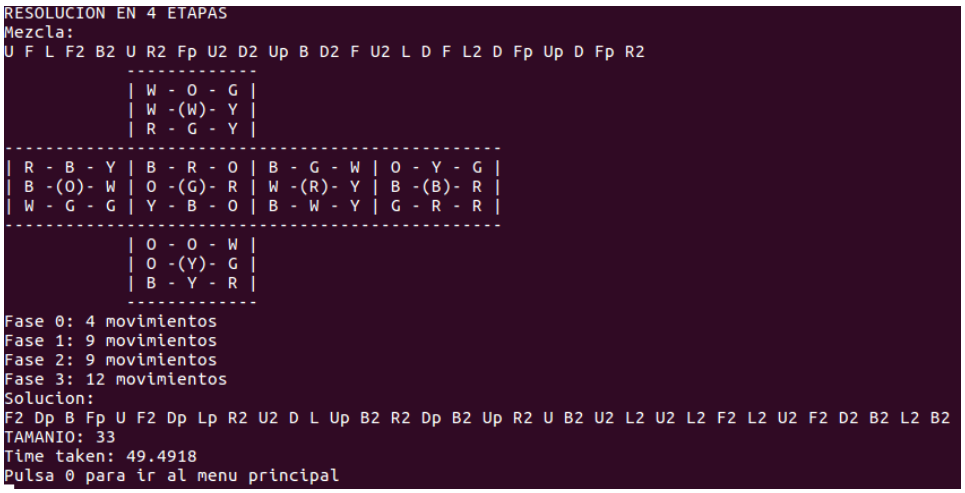
Para empezar a usar el programa debemos compilar y enlazar todo ejecutando la orden **make** en la terminal. Esto nos generará tres ejecutables: **Anchura222**, **Rubik222** y **Rubik333**. Para ejecutar cada ejecutable habrá que escribir en la terminal: **./**«nombreDelEjecutable». Los dos primeros ejecutables nombrados no tendrán funcionalidad, tan solo habrá que ejecutarlos. Esto es, puesto que el 2x2x2 era solo un paso previo para comprobar la viabilidad del problema para el 3x3x3.

```
MENU PRINCIPAL

1 - Generando una mezcla aleatoria
2 - Partiendo de un cubo resuelto e intercambiando las pegatinas
```

Aquí, podemos observar el menú principal, que tan solo contiene 2 opciones. La primera de ellas nos genera una mezcla aleatoria para nuestro rompecabezas.

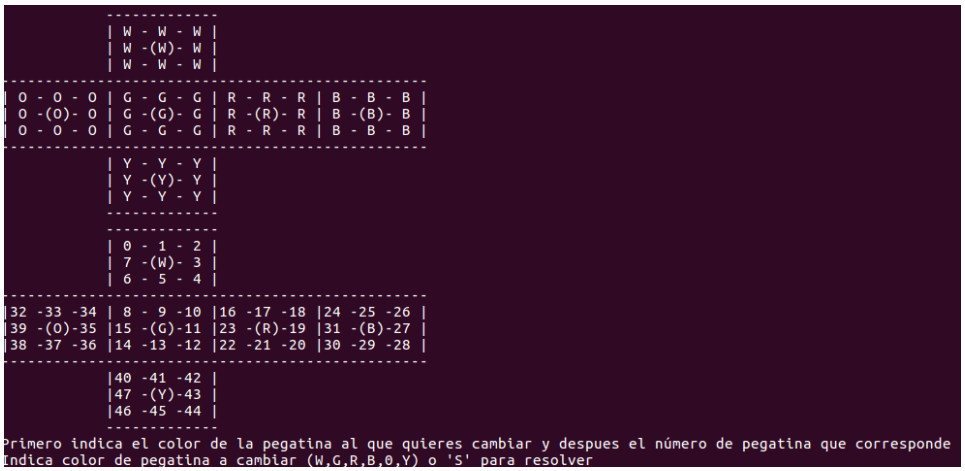
```
Generando mezcla aleatoria
Mezcla:
Rp U2 Dp Up R2 Fp B2 R2 Dp B D L Up Rp Up D R U2 Dp Fp U2 D2 Up D U2
RESOLUCION DEL CUBO

Memetico:
1 - Resolucion en 4 etapas
2 - Resolucion en 3 etapas
Genetico:
3 - Resolucion en 4 etapas
Busqueda Multiarranque:
4 - Resolucion en 4 etapas
0 - Ir atras
```

Como vemos, nos ha generado una mezcla aleatoria de movimientos. Esta mezcla la tenemos que empezar con el Cubo de Rubik resuelto. Después se nos ofrecen 4 opciones (las 4 metaheurísticas que han funcionado) para aplicar una de ellas en su resolución. Por ejemplo, al pulsar «1», pulsar la tecla «Enter» y esperar a que nuestra metaheurística actúe, nos lleva a:




En la que podemos ver la posición inicial del cubo tras la mezcla (posicionando el centro blanco en la cara de arriba y en centro verde en la cara que tenemos enfrente). El número de movimientos requeridos por cada paso de algoritmos y la secuencia que resuelve el cubo. Se muestra también el tamaño de la solución y el tiempo usado.

En la opción número 2 del menú inicial: «Partiendo del cubo resuelto e intercambiando pegatinas» tenemos el siguiente menú:




En el que nos dan la posición de un Cubo de Rubik resuelto y deberemos ir cambiando pegatinas hasta la posición deseada. Esto nos puede ayudar para ciertas configuraciones del cubo donde nos falta poco para resolverlo. Para ir cambiando pegatinas deberemos indicar en primer lugar, el color de la pegatina a cambiar, siendo:

- **W**: white, blanco.
- **Y**: yellow, amarillo.

- **G**: green, verde.
- **B**: blue, azul.
- **R**: red, rojo.
- **O**: orange, naranja.

Y después el número asociado a la posición de la pegatina que queremos cambiar. Tras este cambio se nos mostrará el cubo transformado y podremos seguir cambiando pegatinas. Cuando pulsemos la tecla «S» al elegir color de pegatina, nos mostrará el menú anterior donde podemos elegir el algoritmo a usar. Y después, actuará de la misma forma que hemos explicado antes.

# Capítulo 8

# Software

Para la realización de este proyecto he usado el lenguaje de programación C++. No se trata de una elección al azar, sino que está meditada. Para este problema no era imprescindible una interfaz gráfica. Se trata de un problema de experimentación en el que su objetivo es aplicar Metaheurísticas al Cubo de Rubik. Lo que requería este proyecto era de la mayor eficiencia posible, dado que el uso de Metaheurísticas se sostiene en el tiempo de computo. De ahí, que haya usado C++, un lenguaje de programación muy eficiente y con el que tengo bastante afinidad.

También he utilizado la librería STL (Standard Templaty Library) en ciertas fracciones de mi código. Puesto que esta libreria ofrece algunas estructuras de datos muy eficientes para el uso común. Por ejemplo, en la Búsqueda en Anchura he usado una cola («queue») y en los Algoritmos Genéticos me he servido de los multimaps, para tener ordenada a la población. Esta librería nos favorece mucho en factores como la eficiencia de nuestro código, si se saben usar. Además, nos ahorra mucho trabajo como cualquier librería.

Un aspecto bastante importante pero que parece insignificante es el uso adecuado de las estructuras de control. Debido a la eficiencia que requería este proyecto, he tenido que prestar especial atención a las estructuras de control. En las estructuras **IF-ELSE**, agrupar las sentencias en forma de árbol puede hacerte ganar mucho tiempo. Aunque si se trata de una estructura de árbol con muchas ramificaciones, puede no ayudarte. O, el situar en el primer «if» la opción más probable, también puede hacernos ahorrar gran cantidad de tiempo. Estos detalles tan simples pero a la par tan relevantes en una práctica que solicita tantas iteraciones sobre los mismos métodos, deben tratarse con gran cuidado. He reducido el tiempo de ejecución, aproximadamente, a menos de la mitad atendiendo a estos matices.

# Capítulo 9

# Resultados y análisis

Las Metaheurísticas necesitan trabajar con cromosomas del mismo tamaño para este problema. Las secuencias deben ser de igual tamaño y la idea es ir agrandando la longitud si no encuentra solución. Es decir, partimos con una secuencia de longitud $L$ y si no encuentra solución probaremos con $L + 1$. De esta forma, vamos iterando hasta que una $L$ sea suficiente para encontrar la solución. En mi caso, he usado las siguientes longitudes inciales cuando particiono en 4 pasos:

- **Paso 1**: $L = 2$.
- **Paso 2**: $L = 6$.
- **Paso 3**: $L = 9$.
- **Paso 4**: $L = 8$.

Y para una división en 3 pasos:

- **Paso 1**: $L = 9$.
- **Paso 2**: $L = 9$.
- **Paso 3**: $L = 8$.

Para todos los resultados he usado las mismas semillas y en una cantidad de 50 resultados. Las semillas son: $S = (1, 2, 3, 4, 5, ..., 48, 49, 50)$ para cada media que he hecho.

## 9.1. 2x2x2

Para el 2x2x2 solo puedo hacer la comparación entre Búsqueda en Anchura y Búsqueda Local Multiarranque:

| | **Búsqueda en Anchura** | **BL Multiarranque** |
|---|---|---|
| **Tamaño medio** | 8.26 | 8.74 |
| **Tiempo medio** | 1.28 | 6.79 |

Tengo que decir que, en la Búsqueda Local Multiarranque comienza explorando soluciones de 11 movimientos y después va disminuyendo en uno. Además, el bucle termina cuando itera sobre secuencias de $N$ movimientos y no encuentra solución para ese $N$ dadas 100000 iteraciones. Con esto quiero decir que la Búsqueda Local Multiarranque no emplea un tiempo de ejecución tan superior al de la Búsqueda en Anchura. Solo que la Búsqueda en Anchura explora soluciones de $N$ mayor que el óptimo y también gasta unos 3 o 4 segundos de su tiempo en buscar sin encontrar nada. En cuanto al tamaño de solución si que estan muy cerca y recordemos que la Búsqueda en Anchura siempre encuentra el óptimo. Así que para este puzzle las metaheurísticas funcionan muy bien.

## 9.2. 3x3x3

Búsqueda Local Multiarranque La búsqueda Local Multiarranque ha obtenido unos resultados muy buenos en proporción a su complejidad. Recordemos que esta metaheurística consistía en realizar búsquedas locales comenzando desde distintos puntos del espacio de soluciones hasta encontrar la solución. Se trata de la metaheurística más visual de todas, ya que no es muy complejo, imaginarse un terreno con muchas montañas. En el que tiras muchas bolas desde distintos sitios y el objetivo es que una bola se quede en el punto menos elevado del terreno. Donde cae la bola sería el inicio de la Búsqueda Local y su rodamiento hasta su parada el proceso de búsqueda. Cuesta creer que esta metaheurística no funcione bien en un problema como este, que no es el típico para abordarlo mediante Metaheurísticas. Esta metaheurística es la que debe funcionar lo suficientemente bien en cualquier problema, sin ser la que mejor resultados vaya a dar.

Aquí, muestro los resultados obtenidos con distintos parámetros:

| | | | |
|---|---|---|---|
| **Iteraciones P1** | 30000 | 60000 | 100000 |
| **Iteraciones P2** | 100000 | 200000 | 200000 |
| **Iteraciones P3** | 100000 | 200000 | 300000 |
| **Iteraciones P4** | 100000 | 200000 | 300000 |
| **Iteraciones BL** | 200 | 100 | 50 |
| **Tamaño medio** | 35.96 | 35.34 | 34.78 |
| **Tiempo medio** | 112.62 | 110.82 | 106.701 |
| **Mínimo** | 28 | 27 | 27 |

Siendo:

- Iteraciones P1, P2, P3 y P4: número iteraciones que la búsqueda local hará para un determinado tamaño de secuencia. Al llegar a este número incrementará en 1 la longitud de la secuencia.
- Iteraciones BL: número de iteraciones que realiza la Búsqueda Local.
- Tamaño medio: longitud media de la secuencia de movimientos que resuelve el Cubo de Rubik.
- Tiempo medio: tiempo de ejecución medio para encontrar la solución.
- Mínimo: mínimo valor en el tamaño de una secuencia.

Como podemos apreciar, a medida que voy decrementando el número de iteraciones de la Búsqueda Local voy aumentando el número de iteraciones en cada paso. Esto consigue un equilibrio en el tiempo medio para poder comparar los distintos algoritmos con una mayor facilidad. Todos los tamaños de solución estan en un rango pequeño. El que mejor resultados ha conseguido tanto en tamaño como en tiempo es el tercer algoritmo. Que es el que más explora y menos explota. Y los peores resultados son los del primer algoritmo que hace lo inverso al anterior, explora menos y explota más.

### 9.2.1. Algorimo Genético

El Algoritmo Genético es la metaheurística que más se ha usado para este problema. Y como ya veremos, es la que peor resultados me ha dado de todas. Aunque he de decir, que no esperaba que funcionara muy bien para este problema. Puesto que al cruzar dos secuencias van a formar dos hijas muy dispares. Pero bueno, veamos los resultados:

| | AG1 | AG2 | AG3 |
|---|---|---|---|
| **Generaciones P1** | 3000 | 3000 | 3000 |
| **Generaciones P2** | 15000 | 15000 | 15000 |
| **Generaciones P3** | 15000 | 20000 | 20000 |
| **Generaciones P4** | 10000 | 15000 | 15000 |
| **Tamaño población** | 100 | 100 | 100 |
| **Mutaciones** | 40 | 40 | 20 |
| **Elitismo** | 2 | 2 | 5 |
| **Tamaño medio** | 43.02 | 41.04 | 43.64 |
| **Tiempo medio** | 90.50 | 100.87 | 118.51 |
| **Mínimo** | 30 | 32 | 33 |

Siendo:

- Tamaño población: el número de cromosomas.
- Mutaciones: el número de cromosomas que muta en cada generación.
- Elitismo: la cantidad cromosomas (los mejores) que se mantienen en la siguiente generación.

Podemos observar como el tiempo medio es muy parecido al de la Búsqueda Local Multiarranque. Sin embargo, el tamaño medio de solución es, aproximadamente, 7 movimientos mayor. También se puede apreciar como era necesario mantener un número de mutaciones bastante alto para encontrar soluciones más cortas.

El haber obtenido los peores resultados en el Algoritmo Genético me hace reflexionar sobre varias cosas. La primera de ellas, es que no tengo con quien comparar los otros algoritmos, así que puede que las otras metaheurísticas sean mejor para este problema que el Algoritmo Genético. La segunda, es que no domino los Algoritmos Evolutivos y estos son muy complejos. Los algoritmos Genéticos seguramente sea la metaheurística que más variedad te ofrece a la hora de implementarla. Puedes probar distintos tipos de cruces, distintos tipos de mutaciones, distintas formas de cruzar a los cromosomas,... Todo esto, con cierta experiencia, supongo que debes saber como actúa cada parámetro y puedes ir orientandolo a una mejor optimización. Yo he probado varios tipos de cruce, siendo el mejor el primero que pensé (sin ser realmente bueno). Y he hecho cambios en distintos parámetros, pero no he conseguido buenos resultados.

### 9.2.2. Algoritmo Memético en 4 fases

Este algoritmo combina las dos metaheurísticas de las que he hablado anteriormente. Hay distintas formas de aplicar la Búsqueda Local a los Algoritmos Genéticos. Podemos aplicar la Búsqueda Local solo a los mejores

individuos, podemos aplicarla cada pocas iteraciones o cada muchas, podemos aplicarla a todos los individuos, podemos aplicar diferentes tipos de Búsqueda Local de acuerdo al tipo de cromosoma,... Tras varios intentos, con pocas soluciones la mejor opción era aplicar Búsqueda Local a toda la población. Esto se debe a que cualquier secuencia con un valor heurístico no demasiado alto puede ser solución tras algunas mutaciones. Los resultados obtenidos han sido:

| | AM1 | AM2 | AM3 | AM4 | AM5 |
|---|---|---|---|---|---|
| **Generaciones P1** | 3000 | 3000 | 3000 | 3000 | 3000 |
| **Generaciones P2** | 6000 | 6000 | 9000 | 9000 | 9000 |
| **Generaciones P3** | 8000 | 8000 | 12000 | 12000 | 12000 |
| **Generaciones P4** | 5000 | 5000 | 12000 | 12000 | 12000 |
| **Tamaño población** | 100 | 100 | 100 | 100 | 100 |
| **Mutaciones** | 40 | 40 | 40 | 20 | 40 |
| **Elitismo** | 2 | 2 | 2 | 5 | 2 |
| **Iteraciones BL** | 50 | 100 | 100 | 100 | 100 |
| **Generaciones BL** | 5 | 10 | 10 | 10 | 3 |
| **Tamaño medio** | 35.32 | 35.26 | 34.88 | 35.86 | 33.7 |
| **Tiempo medio** | 75.25 | 72.38 | 111.30 | 122.56 | 236.04 |
| **Mínimo** | 27 | 28 | 28 | 31 | 28 |

Siendo:

- Generaciones BL: cada cuantas generaciones se ejecuta sobre toda la población la Búsqueda Local.
- Iteraciones BL: la cantidad de iteraciones que realiza la Búsqueda Local.

Este algoritmo mejora en tiempo a la Búsqueda Local Multiarranque si queremos obtener soluciones de unos 35 movimientos (AM1 y AM2). Aunque para encontrar soluciones menores de 35 necesita prácticamente el mismo tiempo que la Búsqueda Local Multiarranque. Esto se debe a que converge antes, puesto que parte de secuencias de mediana calidad. Quizá solo necesitemos unos 40 segundos para encontrar una soluciones por debajo de los 36 movimientos. Con lo cual, este algoritmo nos proporciona rapidez a la hora de encontrar una «buena» solución.

Para los AM3, AM4 y AM5 tenemos mayor número de iteraciones en cada paso excepto para el primero. Esta decisión se debe al bajo número de movimientos que necesita esta fase. El espacio de soluciones será muy pequeño, y no requerirá de demasiadas iteraciones. El AM4 es el que peores resultados obtiene ya que realizamos tan solo 20 mutaciones. Para los Algoritmos Genéticos este suele ser un porcentaje común, pero depende de cada

problema. Para este lo ideal ha sido realizar un 40 % de mutaciones y hemos obtenido buenos resultados. El AM3 consigue bajar de 35 movimientos en menos de 2 minutos. Y el AM5 resuelve el cubo en una media de 33.7 movimientos en meos de 4 minutos. Menos de 34 movimientos para resolver el Cubo de Rubik es un gran resultado.

### 9.2.3. Algoritmo Memético en 3 fases

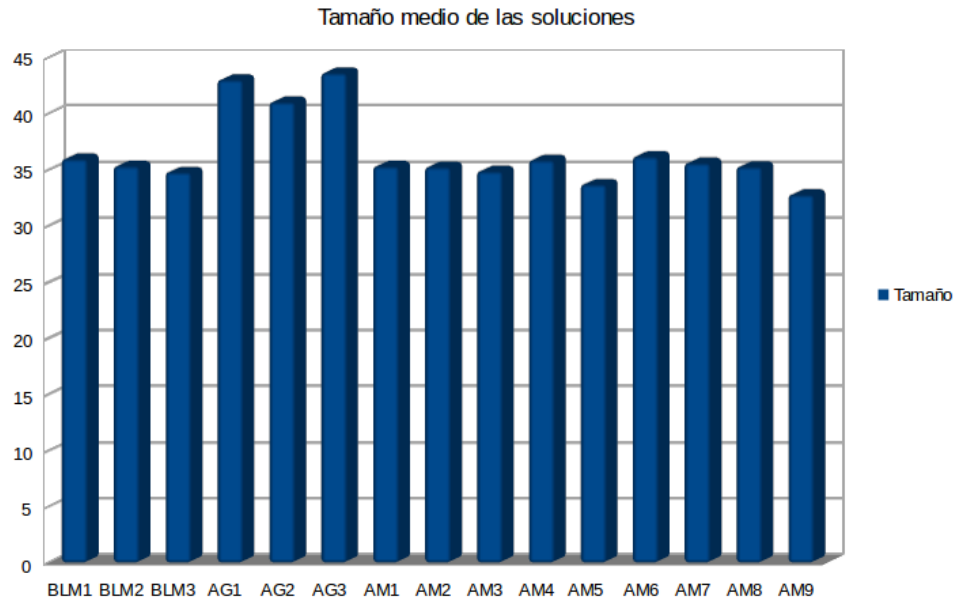
Y por último, los Algoritmos Meméticos han sido los únicos en proporcionar una solución en 3 fases sin que consumamos demasiado tiempo. Con lo cual, se convierte en la metaheurística más apropiada para encontrar secuencias que solucionen este rompecabezas. Estos han sido los resultados obtenidos:

| | AM6 | AM7 | AM8 | AM9 |
|---|---|---|---|---|
| **Generaciones P1** | 3000 | 3000 | 3000 | 3000 |
| **Generaciones P2** | 6000 | 6000 | 9000 | 9000 |
| **Generaciones P3** | 8000 | 8000 | 12000 | 12000 |
| **Generaciones P4** | 5000 | 5000 | 12000 | 12000 |
| **Tamaño población** | 100 | 100 | 100 | 100 |
| **Mutaciones** | 40 | 40 | 40 | 40 |
| **Elitismo** | 2 | 2 | 2 | 2 |
| **Iteraciones BL** | 50 | 100 | 100 | 100 |
| **Generaciones BL** | 5 | 10 | 10 | 3 |
| **Tamaño medio** | 36.18 | 35.6 | 35.28 | 32.78 |
| **Tiempo medio** | 189.03 | 162.39 | 214.36 | 415.94 |
| **Mínimo** | 24 | 29 | 28 | 24 |

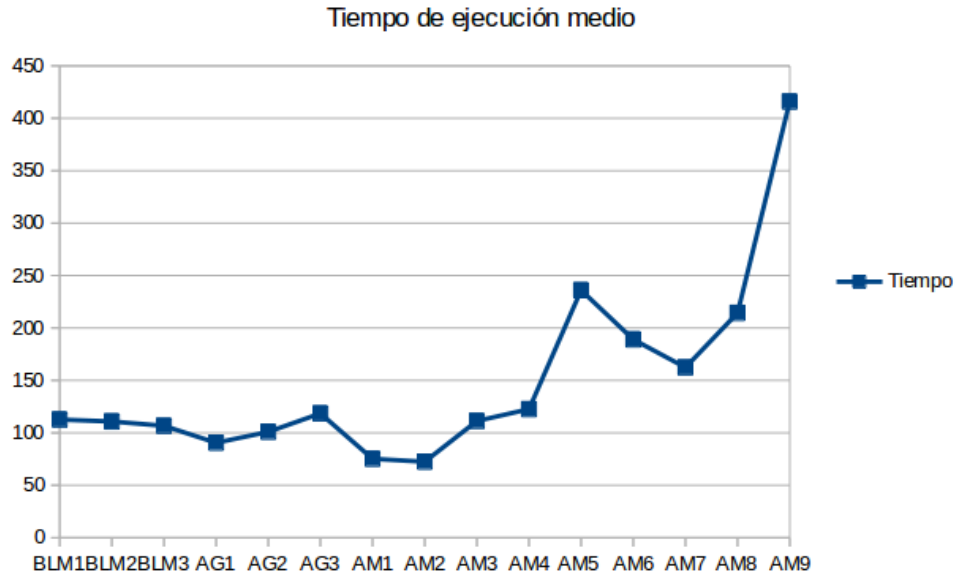
A pesar de ser la única metaheurística capaz de enfrentarse a resolver el Cubo de Rubik en las 3 etapas ya mencionadas, se ve un incremento en el tiempo de ejecución considerable. Para encontrar soluciones cercanas a 35 movimientos necesitamos más de 3 mínutos y medio. Aunque esta división en 3 pasos es la que nos puede optimizar más el número de movimientos (exceptuando el Algoritmo Kociemba). Lo vemos en el AM4, que consigue bajar de 33 movimientos, aunque necesita casi 7 minutos para ello. Estos son los mejores resultados en cuanto a número de movimientos obtenidos puesto que el Algoritmo Kociemba es intratable (en tiempo razonable) en su segundo paso.

### 9.2.4. Comparación

Vamos a empezar comparando el tamaño medio de todos los algoritmos:



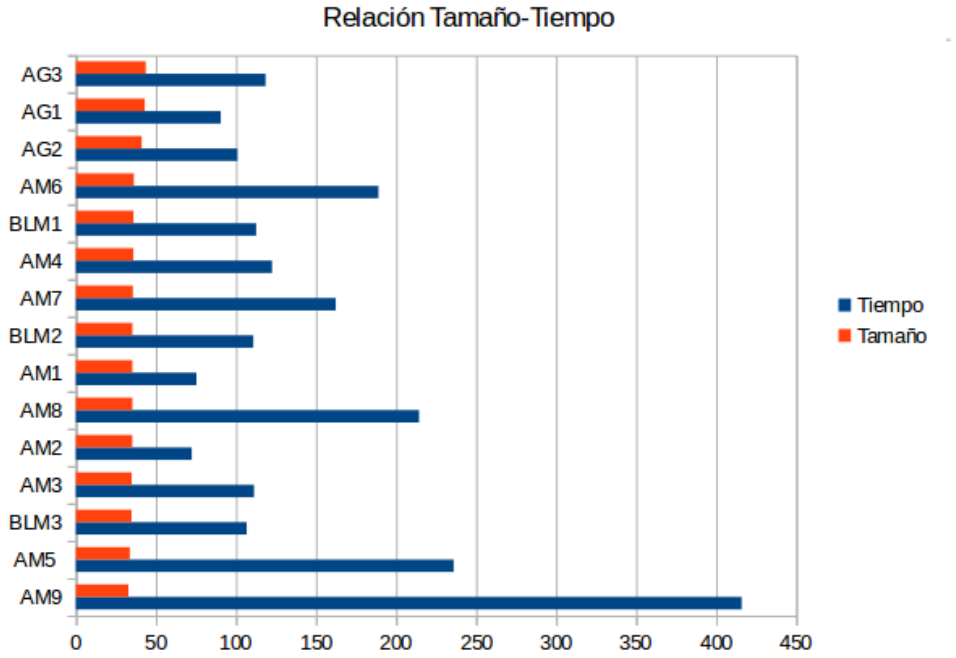

Como podemos visualizar, exceptuando los Algoritmos Genéticos que están por encima de los 40 movimientos, los demás rondan los 35 movimientos. En la Búsqueda Local Multiarranque no hay demasiada variedad, puesto que los tiempo de ejecución eran muy similares. Pero en los Algoritmos Meméticos los tamaños están entre 32.78 y 36.18. Los dos mejores resultados (AM5 y AM9) se deben a que hemos realizado Búsquedas Locales cada 3 generaciones en ambos casos. Aunque como veremos en la siguiente gráfica, esto consume mucho tiempo.




Hasta el AM4 las soluciones rondan los 100 segundos de ejecución, siendo

el AM1 y el AM2 los que menos tiempo tardan en encontrar solución. Este rango de soluciones pertenece a soluciones en 4 fases. El AM5 también pertenece, pero lo forzamos mediante el uso de Búsqueda Local cada pocas generaciones. En el resto de algoritmos resolvemos el cubo en 3 pasos y esto nos supone un incremento en el tiempo total.

Veamos la relación entre tiempo y tamaño, con los tamaños ordenados de forma decreciente:




El que mejor relación tamaño-tiempo presenta es el AM2 seguido de cerca del AM1. Mientras que el AM9 es el que muestra peor relación tamaño-tiempo. Y esto será cada vez más notable según le incluyamos mayor número de iteraciones. Pero para conseguir número bajo de movimientos hay que forzar al algoritmo con un gran número de iteraciones en cada paso.

# Capítulo 10

# Conclusiones y trabajos futuros

Hemos cumplido todos los objetivos principales que requería este proyecto excepto el empleo del algoritmo de la Colonia de Hormigas. Esto se debe a que pensé que esta metaheuristica funcionaría ya que va explorando la solución paso a paso. Para aplicar este algoritmo, debemos tener un grafo, que en este caso podría ser el de los estados del Cubo de Rubik, como nodos, y los movimientos posibles a cada estado, como enlaces. En este grafo, las hormigas irían explorando distintos caminos (que serían secuencias del 3x3x3) hasta llegar a la solución. El problema que tiene este procedimiento, y que no vi al plantearlo de inicio, es que cuando tu pasas de un estado a otro en un Cubo de Rubik es muy difícil saber si estas más cerca o más lejos de la solución. Como ya comenté, encontrar una heurística que nos pueda aproximar a cuantos movimientos estamos de la solución es muy complejo. Y cuando digo aproximar, me refiero a quedarnos relativamente cerca del número mínimo de movimientos a la solución. Con lo cual, las hormigas irían recorriendo el grafo sin tener una heurística que aporte demasiada información sobre el destino a llegar. Este tipo de metaheurísticas se suelen usar en la búsqueda de caminos. Para encontrar un camino corto de $A$ a $B$ en un grafo de distancias, por ejemplo.

Aunque hay que hablar de lo logrado en este proyecto también, como son el resto de objetivos planteados. Resolver el 2x2x2 con Metaheurísticas y sin dividirlo en fases está conseguido. Además, me he aproximado mucho al óptimo y el tiempo de ejecución. Teniendo en cuenta cómo he búscado la solución, no es mucho mayor que el de la Búsqueda en Anchura. Después teniamos el objetivo de resolver el 3x3x3 usando Metaheurísticas aunque para ello hubiese que dividir el Cubo de Rubik en fases. Este objetivo está cumplido con Búsqueda Local Multiarranque, Algoritmos Genéticos y Algoritmos Meméticos. De ahí, que el objetivo 3 también lo hayamos alcanzado. Y, por último, no aportarle a los algoritmos ninguna secuencia que les pueda

ayudar a gestionar alguna situación. Las metaheuristicas que he implementado solo manejan cadenas de movimientos y saben como quedará el Cubo de Rubik después de aplicar cierta secuencia de movimientos. Este es el conocimiento limitado que tienen las metaheurísticas que yo he implementado. Al contrario que otras que he visto, que en el concepto de mutaciones ejecutaban secuencias de movimientos que cambiaran pocas piezas del cubo. Secuencias de este tipo pueden ser:

$$R\ U\ R'\ U\ R\ U2\ R'\ L'\ U'\ L\ U'\ L'\ U2\ L$$

que tan solo cambia la orientación de 2 esquinas en el cubo y mantiene en su sitio a todas las piezas.

En cuanto a los resutados obtenidos, están mejor de lo que me esperaba desde un principio. Entro dentro del grupo medio de los que se han enfrentado a este problema usando Metaheurísticas, si vemos el apartado «Antecedentes». Hablo de número de movimientos necesarios para resolver el Cubo de Rubik, que era lo que había que optimizar. Por tener una referencia humana, hay una modalidad del Cubo de Rubik en la que te proporcionan una mezcla y tu debes encontrar una solución con el menor número de movimientos. Para este cometido, tienes un tiempo de 1 hora y hojas de papel donde deberás ir apuntando todo lo que necesites. El mejor del mundo en esta modalidad puede conseguir una media de 27-28 movimientos en 100 intentos. Si comparamos el tiempo, un humano puede completar en 1 hora el Cubo de Rubik en menos movimientos que una computadora en 4 minutos. Esto no parece que sea difícil, ¿verdad?. La gran diferencia es que el humano para conseguir estos resultados se conoce entre 300 y 1000 secuencias de memoria. Después, mediante teoría del Cubo de Rubik puede realizar ciertas secuencias que permutan solo y de la manera que el quiera 3 esquinas o 3 aristas del Cubo de Rubik. Esto nos supondría una cantidad de más de 1000 secuencias (conmutadores) que el humano especializado puede generar sin pensar más de 2 segundos. Pero no solo eso, dado que puede permutar las 3 piezas que quiera con una secuencia, esta secuencia la podrá incluir en cualquier parte de su solución permitiendo cancelaciones de movimientos. A parte, debemos de tener en cuenta el conocimiento que posee un experto en esta modalidad. Una gran variedad de métodos y aprendizaje basado en la experiencia. Y aunque es muy complejo encontrar una heurística para saber lo cerca que estamos de la solución, un experto en esta modalidad puede ver con cierta facilidad si está a unos 5 o 6 movimientos de la solución. Entonces, no es apropiado comparar a un humano con una metaheurística para este problema.

Ya hablé de esto en el apartado «Heurística» del 3x3x3, pero me parece un aspecto a estudiar que podría ser bastante beneficioso. Aplicar Machine Learning a encontrar una heurística más apropiada para el Cubo de Rubik mediante Aprendizaje Supervisado. Habría, eso sí, que limitar el tiempo

de computo de esta predicción del número de movimientos mínimos a la solución. Esto se podría reducir seleccionando las variables más relevantes y menos costosas para su predicción.

En el aspecto de Ingeniería de Conocimiento también se podría aportar beneficios a este problema. Incluir el conocimiento de un experto nos ahorraría búsquedas en espacios que no nos llevarían a nada. Nos aportaría ciertos atajos y distintas opciones frente a situaciones. Porque el algoritmo Thistlethwaite siempre sigue el mismo procedimiento y hay veces que una sola etapa puede consumir mucho tiempo. Esto se debe a que las etapas anteriores han conducido al 3x3x3 a esta situación. Un procedimiento que se me ocurrió para afrontar esto, pero que no implementé porque iba a gastar mucho tiempo es hacer esto para cada fase del algoritmo Thistlethwaite:

1. Encontrar $N$ secuencias distintas que resuelvan la primera fase.
2. Bucle hasta el paso final.
    *a*) Para cada secuencia de las $N$ que tenemos, encontrar $N$ secuencias que las continúen y que resuelvan la siguiente fase (estas secuencias tienen que ser la concatenación de las de los pasos anteriores).
    *b*) Seleccionar las $N$ secuencias del paso anterior que menos movimientos tengan.

De esta manera, ramificamos y ampliamos el número de opciones, pero siempre seleccionando las mejores para no consumir demasiado tiempo.

A todo esto mencionado, se le podría aplicar el mayor grado de paralalismo e intentar reducir el tiempo de ejecución aprovechando varios procesadores. Ya que se trata de un problema que requiere mucha eficiencia por la gran cantidad de configuraciones posibles que tiene el Cubo de Rubik. Yo he intentado aplicar paralelismo a ciertos bucles y a ciertas porciones de código, pero no he conseguido disminuir el tiempo de búsqueda. Quizá se deba a mi poca práctica con bibliotecas para aplicar paralelismo o a que tal y como he planteado el problema no merezca la pena aplicarlo.

# Capítulo 11

# Bibliografía

Aquí está la bibliografía consultada:


https://ruwix.com/the-rubiks-cube/mathematics-of-the-rubiks-cube-permutation-group/
https://www.jaapsch.net/puzzles/thistle.htm
http://www.human-competitive.org/sites/default/files/borschbach-ppsn-paper.pdf
https://www.jaapsch.net/puzzles/compcube.htm
https://www.cube20.org/
http://www.diva-portal.org/smash/get/diva2:816583/FULLTEXT01.pdf
https://www.ryanheise.com/cube/human_thistlethwaite_algorithm.html
https://www.speedsolving.com/wiki/index.php/Thistlethwaite%27s_algorithm
https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube
http://www.mecs-press.org/ijeme/ijeme-v8-n1/IJEME-V8-N1-1.pdf
https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf
https://www.zunny.com/RUBIK.HTM
https://sci2s.ugr.es/graduateCourses/Metaheuristicas
https://www.francocube.com/cyril/genetic_alg_sol82
https://www.skilled.io/u/swiftsummit/rubik-s-cubes-and-genetic-algorithms-in-swift
https://arxiv.org/pdf/1805.07470.pdf
http://www.human-competitive.org/sites/default/files/borschbach-ppsn-paper.pdf
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.659.9580&rep=rep1&type=pdf
http://grrroux.free.fr/method/Intro.html
https://www.speedsolving.com/wiki/index.php/CFOP_method