



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**



Grado en Ingeniería Informática

Trabajo Fin de Grado

**Diseño y Desarrollo de un Resolvedor
para Cubos de Rubik con Bloqueos**

Autor: Alejandro Soriano Compta
Tutor(a): Julio Mariño Carballo

Madrid, Junio de 2025

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

*Trabajo Fin de Grado
Grado en Ingeniería Informática*

Título: Diseño y Desarrollo de un Resolvedor para Cubos de Rubik con Bloqueos

Junio de 2025

Autor: Alejandro Soriano Compta

Tutor: Julio Mariño Carballo

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software

Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

El cubo de Rubik original se inventó en 1974. Desde entonces han surgido multitud de rompecabezas tridimensionales similares, con variaciones sobre otros poliedros y figuras geométricas, o modificaciones en las formas de giro o colores. Este trabajo consiste en la implementación de un solucionador general para cubos con bloqueos, o *Bandaged Cubes*. Estos cubos se crean uniendo físicamente algunas piezas a un cubo 3x3x3 tradicional, y restringir algunos giros dependiendo de la situación de los bloqueos en cada momento. Debido a que hay multitud de opciones a la hora de seleccionar las piezas que se unen, este solucionador es general, es decir, encuentra soluciones para cualquier combinación de bloques.

Existen solucionadores de cubos tradicionales (por ejemplo, *Cube Explorer* [1]), que, entre otras técnicas, aprovechan su estructura matemática de *grupo no conmutativo*. Tras unir varias piezas en bloques, esta estructura no tiene por qué preservarse, lo que impide utilizar ciertas técnicas de resolución para cubos estándar. Este es un trabajo pionero en su campo, puesto que no existen herramientas similares con un buen desempeño, al menos de dominio público.

El proyecto ha sido implementado principalmente en *Haskell*, y cuenta con una interfaz de visualización programada en la librería *Manim* de *Python* para facilitar su uso. El ámbito principal de este programa es usarlo en cubos muy restringidos, los cuales pueden ser complejos de resolver para un humano. En estos casos, el programa encuentra soluciones óptimas en un tiempo de unos pocos segundos. No obstante, para cubos con pocas o ninguna restricción, puede tardar demasiado tiempo, del orden de horas. Todavía hay margen de mejora en el algoritmo para reducir este tiempo y hallar soluciones en un tiempo razonable para todos los casos.

Utilizando este solucionador, se ha encontrado una solución óptima de 252 movimientos para una posición de un cubo concreto. Algunos usuarios ya afirmaban que este tamaño era óptimo [2], y este solucionador lo corrobora. También se han encontrado algunas posiciones resolubles en 22 movimientos para 2 cubos concretos, lo cual nos muestra que el número máximo necesario para resolver algunas configuraciones de bloqueos puede exceder el del 3x3x3 convencional, que es 20 [3]. Este no es el primer cubo con bloqueos que presenta esta propiedad [4].

Abstract

The original Rubik's Cube was invented in 1974. Since then, many similar three-dimensional puzzles have emerged, featuring variations based on other polyhedra and geometric shapes, or modifications in its turning mechanisms or color schemes. This work focuses on the implementation of a general solver for *Bandaged Cubes*. These puzzles are created by physically attaching certain pieces of a traditional $3 \times 3 \times 3$ cube, limiting the possible moves available depending on the positions of the blocks in every moment. Since there are many possible ways to select which pieces are joined, this is a general solver, which means that it can find solutions for any combination of blocks.

There are solvers for traditional cubes (such as Cube Explorer [1]) that, take advantage of its mathematical structure of *non-commutative group*, among other techniques. However, once several pieces are joined into blocks, this structure is rarely preserved, which prevents the use of certain solving techniques used for standard cubes. This is a pioneering work in its field, as there are no comparable tools of public domain with good performance currently available.

The project has been implemented primarily in *Haskell*, and includes a visualization interface developed using *Python's Manim* library to facilitate its usage. The main purpose of the program is to solve highly constrained cubes, which can be a complex challenge for humans. In such cases, the program finds optimal solutions within a few seconds. However, for cubes with few or no restrictions, computation time can become excessive, on the order of several hours. There is still room for improvement in the algorithm to reduce this time and achieve reasonable performance across all scenarios.

Using this solver, an optimal solution of 252 moves was found for a specific cube configuration. Some users had already suggested that this was the optimal value [2], and this solver confirms it. Additionally, certain positions in two specific bandaged cubes have been solved in just 22 moves, which indicates that the maximum number of moves required to solve some blocked configurations can exceed that of the conventional $3 \times 3 \times 3$ cube, which is 20 [3]. This is not the first bandaged cube that present this property [4].

Tabla de contenidos

1. Introducción	1
1.1. Descripción del problema	1
1.2. Motivación	3
1.3. Lista de objetivos y tareas	5
1.4. Estado del arte	6
1.5. Descripción de la solución	7
1.6. Organización del resto del manuscrito	7
2. Preliminares	9
2.1. Introducción a los Cubos de Rubik	9
2.1.1. Comprendiendo el puzzle	9
2.1.2. Notación del cubo de Rubik	10
2.1.3. Orientación correcta de las piezas	14
2.2. Cubos con bloqueos	16
2.3. Métodos de resolución del cubo de Rubik	16
2.3.1. Métodos humanos	16
2.3.2. Métodos computacionales	17
2.3.3. Cubos con bloqueos	18
2.4. Introducción a la teoría de grupos	20
2.4.1. Definición	20
2.4.2. Grupos de permutaciones	20
2.4.3. Grupo del cubo de Rubik	21
2.4.4. Grupoide de los cubos con bloqueos	21
2.5. Introducción a la programación funcional y <i>Haskell</i>	22
3. Análisis y diseño	25
3.1. Requisitos	25
3.2. Casos de uso	27
3.3. Estructura y división en módulos	27
3.4. Notación de entrada y definición de bloques	31
3.4.1. Compleción de bloques	31
3.5. Salida y visualización	34
3.6. Modelado de los datos	34
3.6.1. Codificación de un cubo con y sin bloqueos	35
3.6.2. Codificación de giros	37
3.7. Motor de búsqueda	39

TABLA DE CONTENIDOS

3.7.1. Reducción del árbol de búsqueda	40
3.7.2. Algoritmo Iterative-Deepening A* (IDA*)	42
3.7.3. Algoritmo de Korf	43
3.7.4. Heurísticas de Korf	43
4. Desarrollo e implementación	51
4.1. Entrada de los datos	51
4.2. Salida	53
4.3. Visualización	54
4.4. Implementación de los datos	57
4.4.1. Cubo de Rubik estándar	57
4.4.2. Giros	58
4.4.3. Cubo con restricciones	60
4.5. Motor de búsqueda	63
4.5.1. Algoritmo de Korf	63
4.5.2. Generación de heurísticas	68
4.6. Testing	69
5. Resultados experimentales	75
5.1. Evaluación del rendimiento	75
5.2. Contribuciones teóricas y demostraciones	79
6. Análisis de impacto	81
7. Conclusiones y trabajo futuro	83
7.1. Conclusiones	83
7.2. Trabajo futuro	83
Bibliografía	87
Anexos	91
A. Cálculo de los factores de ramificación de algunos subgrupos	91
A.1. Dos capas contiguas, $\langle R, U \rangle$	91
A.2. Tres capas que comparten una esquina, $\langle R, F, U \rangle$	92
A.3. Tres capas, de las cuales 2 opuestas, $\langle R, L, U \rangle$	92
A.4. Cuatro capas, de las cuales 2 opuestas, $\langle R, L, F, U \rangle$	94
B. Código	97
B.1. Programas de interacción con el usuario	97
B.2. Módulo Data	98
B.2.1. Fichero Cube.hs	98
B.2.2. Fichero Moves.hs	99
B.2.3. Fichero Bandaged.hs	104
B.2.4. Fichero MathematicalNotation.hs	108
B.3. Módulo de IO-Visualizator	109
B.3.1. Fichero ExpandBlocks.hs	109
B.3.2. Fichero InputBandagedCube.hs	110

TABLA DE CONTENIDOS

B.3.3. Fichero ManimHsConversion.hs	111
B.3.4. Fichero InputCube.hs	114
B.3.5. Fichero Visualizator.hs	118
B.4. Módulo Heuristics	119
B.4.1. Fichero Combinatorics.hs	119
B.4.2. Fichero IndexHeuristics.hs	120
B.4.3. Fichero GenKorfHeuristics.hs	121
B.4.4. Fichero KorfHeuristic.hs	125
B.5. Módulo Search-Engine	125
B.5.1. Fichero Search.hs	125
B.5.2. Fichero MoveGeneration.hs	128
B.5.3. Fichero SolvingStrategies.hs	131

Capítulo 1

Introducción

1.1. Descripción del problema

El proyecto de este trabajo de fin de grado es el diseño e implementación de un software capaz de resolver cubos de Rubik con bloqueos. Este proyecto se mantendrá en un repositorio de mi GitHub: https://github.com/Alexsql/Real_Bandaged_Cube_Explorer. Estos cubos son una variación del cubo de Rubik 3x3 tradicional, donde algunas de sus piezas de unen físicamente. De este modo, algunos de los giros no están permitidos según las posiciones de los bloques en cada momento. Evidentemente, existen multitud de opciones a la hora de seleccionar qué piezas unir y con qué bloqueos configurar el cubo. Por poner algunos ejemplos de cubos con bloqueos, se muestran ilustraciones de los cubos *Bicube*, *Alcatraz* y *Big Block* en las imágenes 1.1, 1.2 y 1.3 respectivamente.

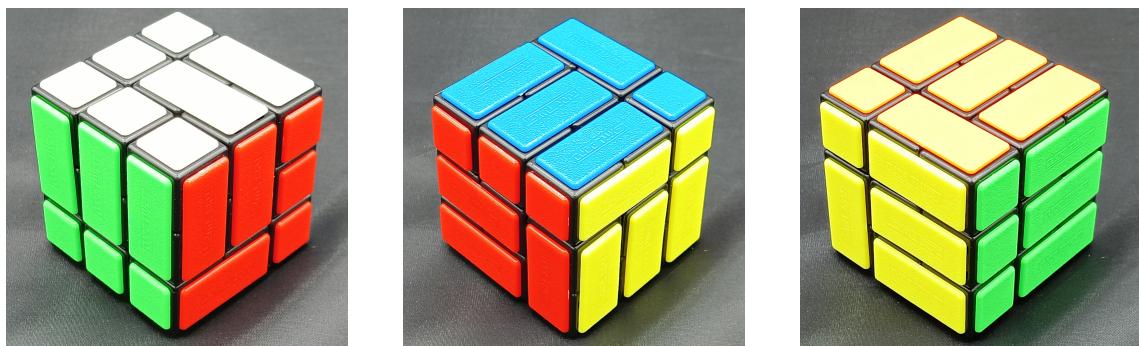


Figura 1.1: *Bicube*

Hay multitud de opciones a la hora de seleccionar los bloqueos, y este trabajo permitirá encontrar una solución para cualquier configuración de ellos. Para crear cubos con bloques configurables, nos será muy útil usar un kit de la marca *Cube twist* (ver figura 1.4). Este cuenta con una base de cubo 3x3 sin colores, sobre el cual podemos colocar piezas planas de colores, sean de 1x1 o de dimensiones superiores. Gracias a esto, podremos colocar los bloques a nuestro gusto, y poder crear cualquier cubo bandaged.

Capítulo 1. Introducción

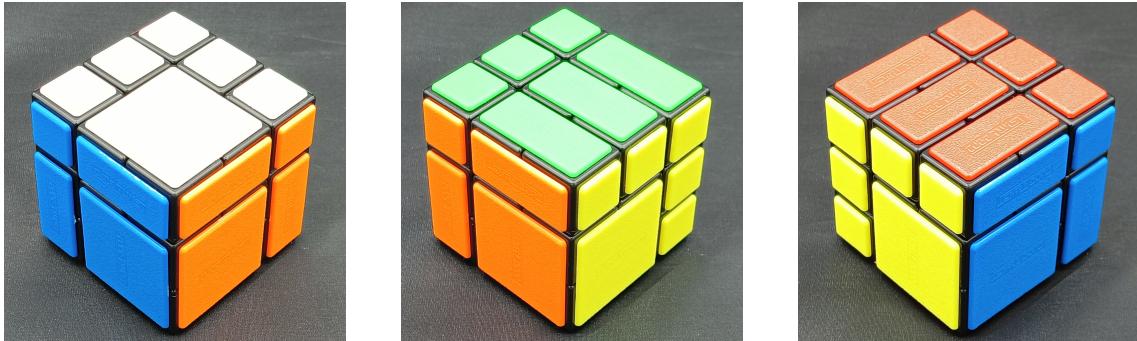


Figura 1.2: Cubo *Alcatraz*



Figura 1.3: Cubo *Big Block*

El objetivo del software de este trabajo será un solucionador general, que pueda resolver cualquier posición un cubo con cualquier configuración de bloques. A día de hoy, no existe ninguna herramienta que realice esta funcionalidad, al menos de dominio público. En este programa, el usuario podrá introducir un estado arbitrario, con un patrón de bloqueos configurable, y el software encontrará, si es posible, los movimientos que ha de realizar para poder resolver el cubo. Esta tarea debe de ser lo más eficiente posible, tanto en longitud de la solución hallada como en recursos computacionales para hallarla (consumo de memoria y tiempo de ejecución principalmente). A su vez, la interfaz de usuario para interactuar con el programa ha de ser lo más sencilla e intuitiva de usar, evitando al máximo que el usuario deba conocer la codificación interna de los datos u otros detalles de implementación.

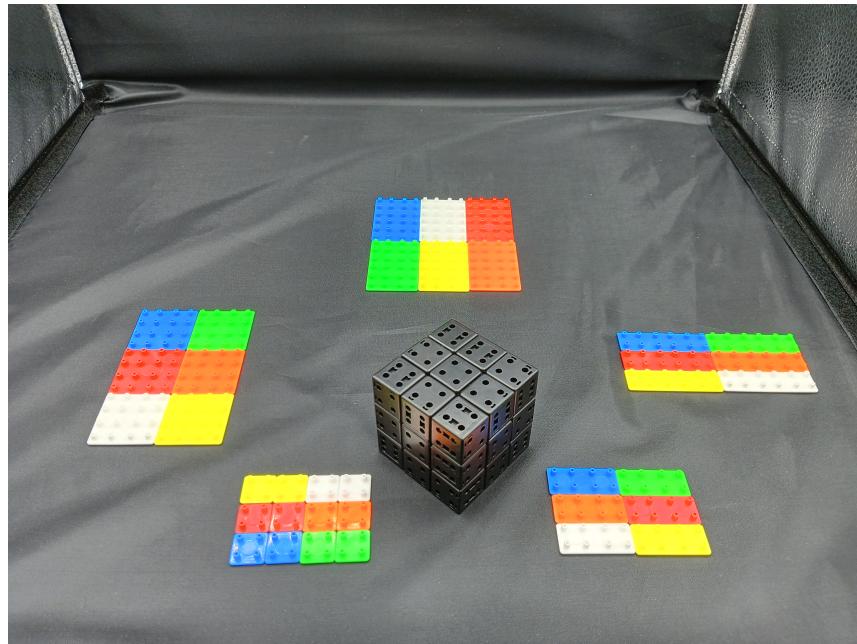


Figura 1.4: Cubo configurable DIY

1.2. Motivación

Los cubos bandaged pueden tener métodos de resolución muy diferentes a los convencionales, en especial aquellos con muchas restricciones. Hay usuarios de internet compartiendo nuevos descubrimientos y formas nuevas de resolver estos cubos. En la imagen 1.5 se muestra el grafo de Cayley según la posición de los bloques del cubo *Bicube Fuse*. Este diagrama fue generado por el usuario *ladislavdubravsky* [5], con el objetivo de encontrar una solución al cubo.

Capítulo 1. Introducción

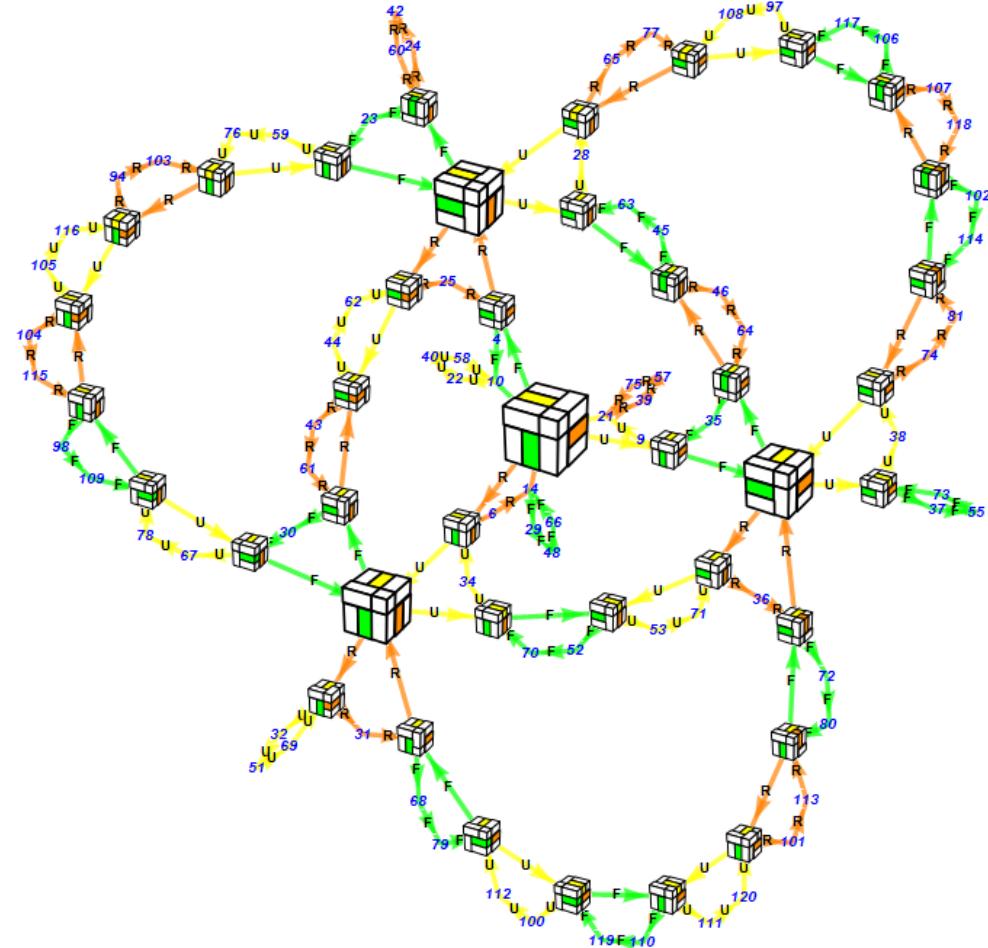


Figura 1.5: Grafo de posiciones del cubo *Bicube Fuse* [5]

Otro usuario que publica contenido online resolviendo cubos con bloqueos es *TheMaoiSha* [6]. Este cuenta con varios videos en su canal de youtube donde muestra estos cubos, además de analizarlos, estudiarlos y resolverlos. En ocasiones, muestra la forma en la que encontró la solución de algunos puzzles (ver imagen 1.6) [7], [2].

1.3. Lista de objetivos y tareas

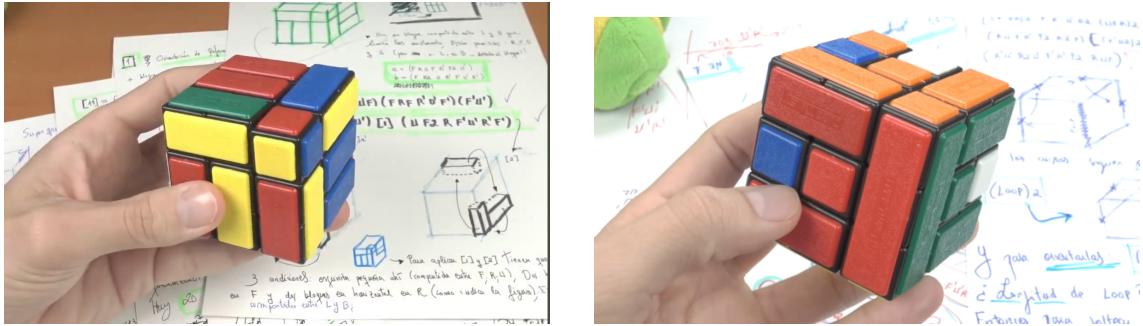


Figura 1.6: El usuario de youtube *TheMaoiSha* mostrando y resolviendo cubos con bloqueos [7], [2]

Algunas formas de resolverlos tratan de adaptar los métodos para cubos existentes, aunque suele valer para aquellos con pocas restricciones. Lo que suele intentarse en cubos bandaged es comprender el estado de geometrías del cubo, averiguar el efecto de algunos ciclos y tratar de hallar secuencias de giros que permuten pocas piezas. Posteriormente, deben de combinarse estas secuencias para ir resolviendo las piezas poco a poco. No obstante, esto no es tan sencillo, puesto que estos grafos pueden tener un tamaño muy extenso, y cada definición de bloques tendrá el suyo. Incluso con estas ideas, sigue siendo un reto complejo. Sigue formando parte de la resolución acceder a la geometría del estado resuelto, saber deducir las secuencias de movimientos de un cubo y saber cuáles utilizar en cada momento. Dada la complejidad de este reto, será muy útil contar con un software capaz de ayudar total o parcialmente a la resolución. Por ejemplo, podrá utilizarse para hallar secuencias de movimientos que permuten pocas piezas. A nivel de programación, realizar este software también es un reto interesante, puesto que adaptar las técnicas de resolución de cubos estándar no es tan sencillo en este caso. A nivel computacional, un cubo de Rubik estándar cuenta con una estructura algebraica de *grupo no conmutativo*, al igual que muchos de los rompecabezas mecánicos tridimensionales. Algunas propiedades de esta estructura suelen ser aprovechadas por los solucionadores de cubos. No obstante, esto suele perderse al añadir bloqueos, lo que obliga a replantearse su solución algorítmica.

1.3. Lista de objetivos y tareas

Los objetivos del proyecto son los siguientes:

- Definir una notación para especificar bloqueos.
- Adaptar notaciones para definir operadores básicos de transformaciones.
- Desarrollar un motor de búsqueda que obtenga operadores básicos.
- Desarrollar un resolvedor de configuraciones arbitrarias a partir de operadores básicos.

Capítulo 1. Introducción

La lista de tareas del trabajo es:

- Modelado simbólico de los datos. Definición e implementación de una notación para representar cubos con bloqueos.
- Definición e implementación del lenguaje de entrada/salida.
- Definición e implementación de operadores básicos de transformación entre estados del cubo.
- Desarrollo de un motor de búsqueda, basado en operadores básicos.
- Publicación del software en github o similar.
- Memoria, documentación y presentación.

1.4. Estado del arte

Existen multitud de solucionadores de cubos de Rubik estándar, pero el más conocido posiblemente sea Cube Explorer, creado por Herbert Kociemba [1]. No obstante, este no permite utilizar cubos con bloqueos. En cuanto a solucionadores de cubos con bloqueos, existen varias herramientas en GitHub con el nombre de “Bandaged Cube Explorer” en honor a Cube Explorer. A pesar de ello, la mayoría no son solucionadores, sino que permiten visualizar los posibles estados de bloqueos o sus espacios de búsqueda. Estas herramientas han sido creadas por los usuarios *ladislavdubravsky* [5], *paulnasca* [8], y *JoshMermel* [9].

Sí que es cierto que existe un software capaz de encontrar soluciones, creada por el usuario *JackTriton* en *Python* [10]. No obstante, este no cuenta con un buen funcionamiento: en primer lugar, el input se introduce como variable en uno de los códigos fuentes, en lugar de utilizar una interfaz que ayude al usuario y posteriormente traduzca a la codificación interna. Es necesario introducir las piezas bloqueadas basándose en la capa a la que pertenecen (R, U, F, L, B ó D). La forma de introducir el estado mezclado que se va a resolver es en base a una mezcla que parte desde el estado resuelto. El input no debería de basarse en la información que necesitas y la que seguramente carezcas, puesto que la solución y la mezcla son secuencias inversas. Por otra parte, el código no contiene revisión ni corrección de errores. En el caso de hacer un giro que rompa algún bloque, el programa lo realiza y accede a posiciones inaccesibles. En cuanto a las búsquedas, se realizan mediante el algoritmo IDDFS, lo cual es mucho menos eficiente que uno heurístico. Cuando llega a una cota máxima y la solución no es encontrada, en lugar de afirmar que no ha encontrado ninguna solución, salta una excepción cuyo mensaje no ayuda al usuario a comprender lo que ocurre. Esto sucede tras un cierto tiempo de búsqueda siempre que se introduce una posición o mezcla imposible o irresoluble. Este trabajo tiene como objetivo resolver estos problemas y superar en funcionamiento y desempeño a todos los programas similares de publicados en la web.

1.5. Descripción de la solución

El solucionador se va a realizar en el lenguaje de programación *Haskell*. Para realizar las búsquedas, se adaptará el algoritmo de Korf [11]. Este es un algoritmo heurístico diseñado para hallar soluciones óptimas del cubo de Rubik 3x3x3. Está basado en el algoritmo IDA* [12], también propuesto por Korf, el cual fue diseñado para resolver problemas de búsqueda sobre espacios de estados de gran tamaño. El algoritmo de Korf utiliza heurísticas basadas en bases de datos de patrones. Por simplicidad, utilizaremos las heurísticas de un cubo sin bloqueos. A este algoritmo, se le añadirá la restricción de generar solamente los giros que no rompan ningún bloque en cada momento, con el objetivo de que encuentre una solución válida y con un número de movimientos óptimo. Además, algunos giros serán descartados estratégicamente para no repetir la exploración de algunos estados ya explorados, y por ello reducir el árbol de búsqueda y mejorar la eficiencia del solucionador. Para facilitar el uso del programa para cualquier usuario no experimentado, también incluirá una pequeña interfaz gráfica escrita en *Python*, que usa la librería *Manim*. La interfaz será capaz de generar un cubo de Rubik convencional por pantalla y una secuencia giros que se ejecuten sobre él. Esto se utilizará para comprobar que el input introducido es correcto, y posteriormente ejecutar los movimientos necesarios para resolver el cubo.

1.6. Organización del resto del manuscrito

En este documento aparecen 7 capítulos. En el capítulo 1 hay una introducción del trabajo. En este hay información como generalidades del proyecto, su utilidad o el estado en el que se encuentra este campo de estudio. A continuación, el capítulo 2, de preliminares, contiene información necesaria para poder comprender el resto del trabajo. Más adelante, en el capítulo de análisis y diseño (3) se detalla el problema a resolver, sus requisitos y su complejidad, y se toman algunas decisiones a nivel de diseño del software, matemático y computacional. Estos conceptos se llevan a la práctica en el siguiente capítulo, de desarrollo e implementación. En este, se detallan las secciones del código más relevantes y se explican algunas de sus decisiones, además de comentar la forma de probar el código. El desempeño de este trabajo será medido en el capítulo 5, de resultados experimentales. En este, se evalúa el rendimiento del programa y se muestran algunos cálculos que se han obtenido con él. Por último, aparecerán el capítulo 6 analizando el posible impacto del trabajo y el capítulo 7, resumiendo lo conseguido en este trabajo y proponiendo algunas ideas con la que continuararlo.

Capítulo 2

Preliminares

2.1. Introducción a los Cubos de Rubik

2.1.1. Comprendiendo el puzzle

Un cubo de Rubik consta de 6 caras, cada una con 9 pegatinas. Cada cara puede rotarse sobre su centro, mediante giros de 90 grados. El objetivo de su resolución es colocarlo de manera que cada cara contenga pegatinas de un único color. El máximo número de movimientos necesario para resolverlo de forma óptima es 20 [3]. Aunque pudiéramos pensar que las pegatinas se mueven de manera independiente, esto no es cierto. Puede ser más conveniente pensarla como 27 pequeños cubos unidos entre sí, pero nuevamente, nos daría a entender que todas estas piezas son intercambiables, lo cual también es erróneo. Existen varios tipos de piezas, que denominamos centros, esquinas (o vértices) y aristas (en inglés se denominan *centers*, *corners* and *edges* respectivamente). Cada uno de los 6 centros contiene un único color. Al realizar movimientos de las capas, cada centro solamente rota sobre sí mismo, sin capacidad de intercambiarse por otro. Por ello, diremos que los centros son fijos, y determinan el color de la cara en su estado resuelto. Cada una de las 8 esquinas del cubo contiene pegatinas de 3 colores diferentes, y de igual manera, cada una de las 12 aristas contiene 2 colores distintos. Por consiguiente, cada arista en una posición puede estar colocada de 2 maneras diferentes (con sus colores intercambiados), a lo que llamamos orientación, y similarmente, cada esquina tiene 3 opciones de orientación en cada posición (ver figura 2.2). Entendiendo los tipos de piezas diferentes del cubo, es fácil ver que al realizar cada giro, cada tipo de pieza termina en una posición del mismo tipo de pieza. La tabla 2.1 resume estas piezas y sus opciones. Cabe destacar que no todas las reordenaciones de piezas son resolubles [13]. Si fijamos la orientación de todas las piezas menos una (sean esquinas o aristas), la orientación de la última quedará determinada por las demás. Por ello, solo un tercio de las orientaciones de esquinas son resolubles, y la mitad de las orientaciones de esquinas. Por último, la paridad del número de intercambios de esquinas y aristas ha de ser igual (es decir, ambos pares o impares).

Capítulo 2. Preliminares

Tipo de pieza	Cantidad de piezas en el cubo	Número de colores (orientaciones)
Centro	6	1
Arista	12	2
Esquina	8	3

Cuadro 2.1: Tipos de piezas en un cubo de Rubik

En la figura 2.1 se ilustra un cubo gris con los tipos de piezas resaltadas en amarillo.

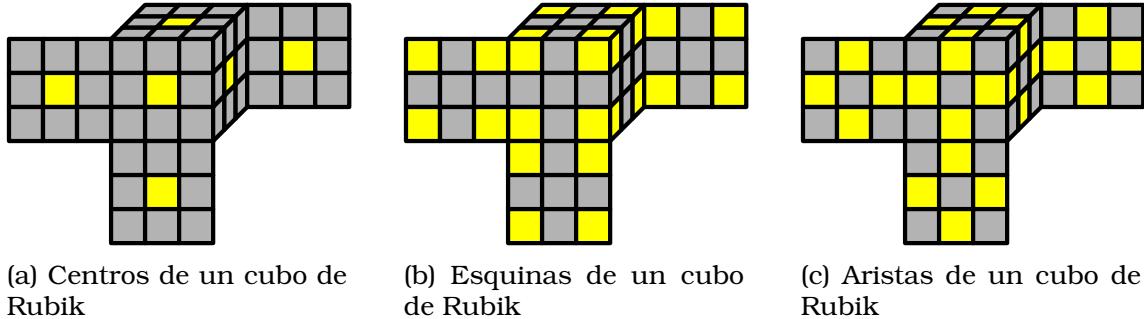


Figura 2.1: Tipos de piezas en un cubo de Rubik

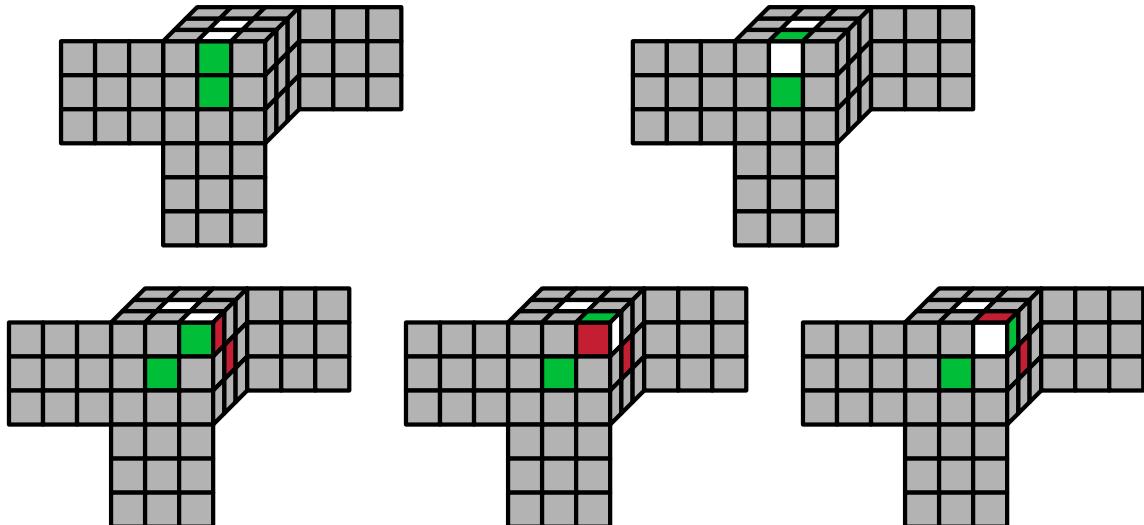


Figura 2.2: Posibles orientaciones de una arista o esquina

2.1.2. Notación del cubo de Rubik

Aunque puede que existan varias notaciones para describir secuencias de movimientos, el estándar de facto es la notación propuesta por David Singmaster en 1981 [14], con algunas variaciones. Esta representa la métrica HTM (*half turn metric*), donde se cuenta como giro cada movimiento de una capa, con independencia del número de grados efectuados. Para ello, se nombra cada cara con su inicial en inglés: **R**ight, **U**p, **F**ront, **L**eft, **D**own y **B**ack. Escribir la inicial indica

2.1. Introducción a los Cubos de Rubik

un movimiento de 90 grados en sentido horario (sentido de las agujas del reloj). La misma inicial seguido de un apóstrofe (se nombra “prima”) indica un giro en sentido antihorario (contrario a las agujas del reloj). Si tras la inicial aparece un 2, indica un giro de 180 grados (el sentido de giro es indiferente). Originalmente, esta notación era más similar a las matemáticas, donde se indicaban R^{-1} para giros en sentido antihorario y R^2 para giros dobles, pero esta notación sufrió modificaciones por motivos de simplicidad y agilidad a la hora de escribirlas. Partiendo desde el estado resuelto ¹, los posibles giros de una capa se muestran en las figuras 2.3 y 2.4. Además, están definidas rotaciones, las cuales consisten en colocar el cubo con otra orientación sin realizar ningún giro. Estas se notan como **x** (similar a R, moviendo todo el cubo), **y** (similar a U) y **z** (análogo a F). También existe notación algo más avanzada que no se incluye en la métrica HTM y por consiguiente no será cubierta en profundidad en este trabajo. Cuando una letra se escriba seguida de una **w**, o la letra de la capa en minúscula, indicará el mismo giro con 2 capas, es decir, moviendo la capa intermedia adyacente simultáneamente. También existe notación para las capas intermedias, representadas como **Middle** (que gira de manera paralela a L), **Equatorial** (paralelo a D) y **Standing** (paralelo a F). En la tabla 2.2 se muestra un resumen de la notación.

¹Por convenio, siempre partimos con la cara verde en frente, y en la cara superior la cara blanca. Esta será nuestra orientación por defecto, salvo que se especifique lo contrario.

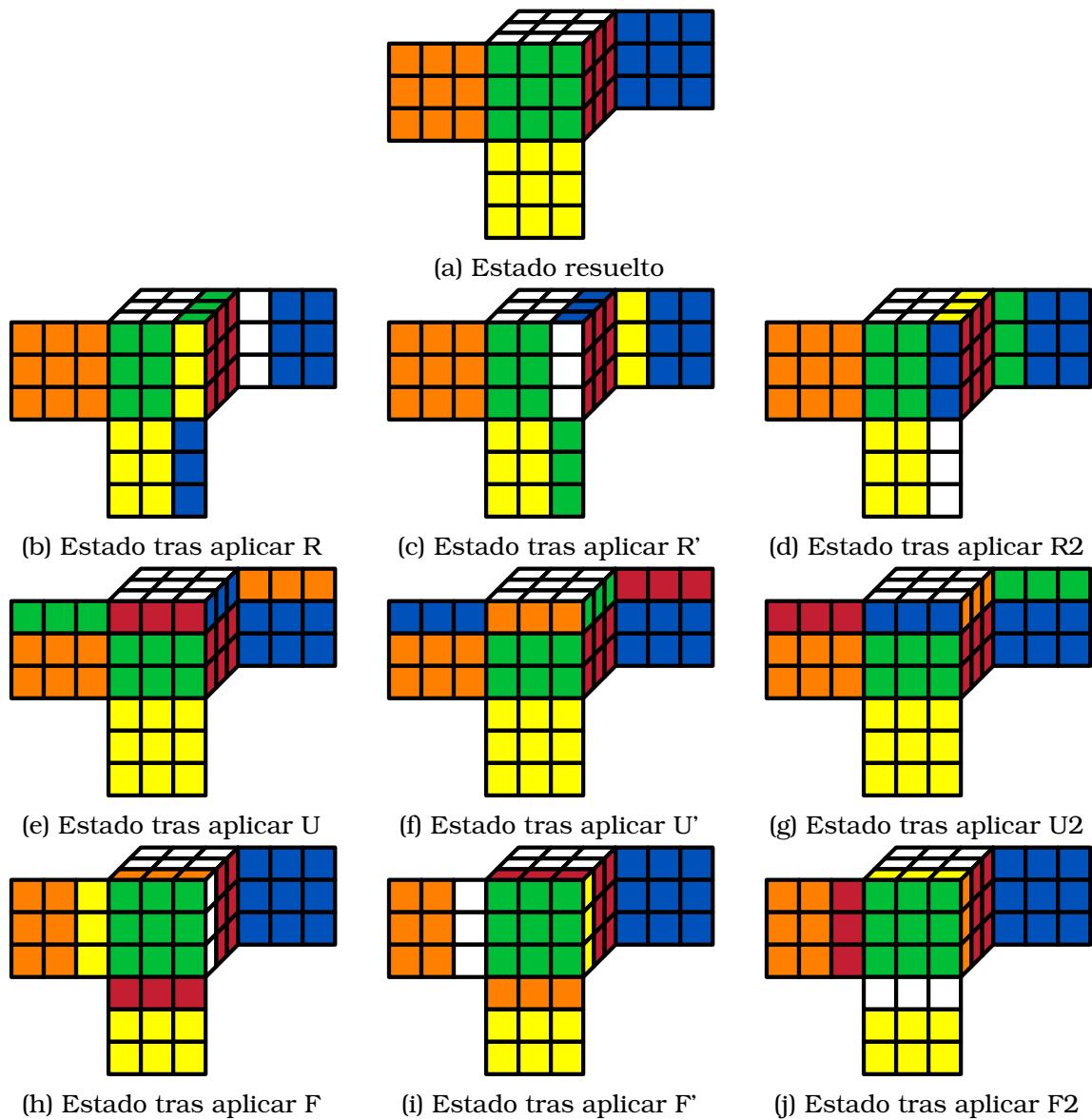


Figura 2.3: Estados tras realizar giros de capas R, U, F

2.1. Introducción a los Cubos de Rubik

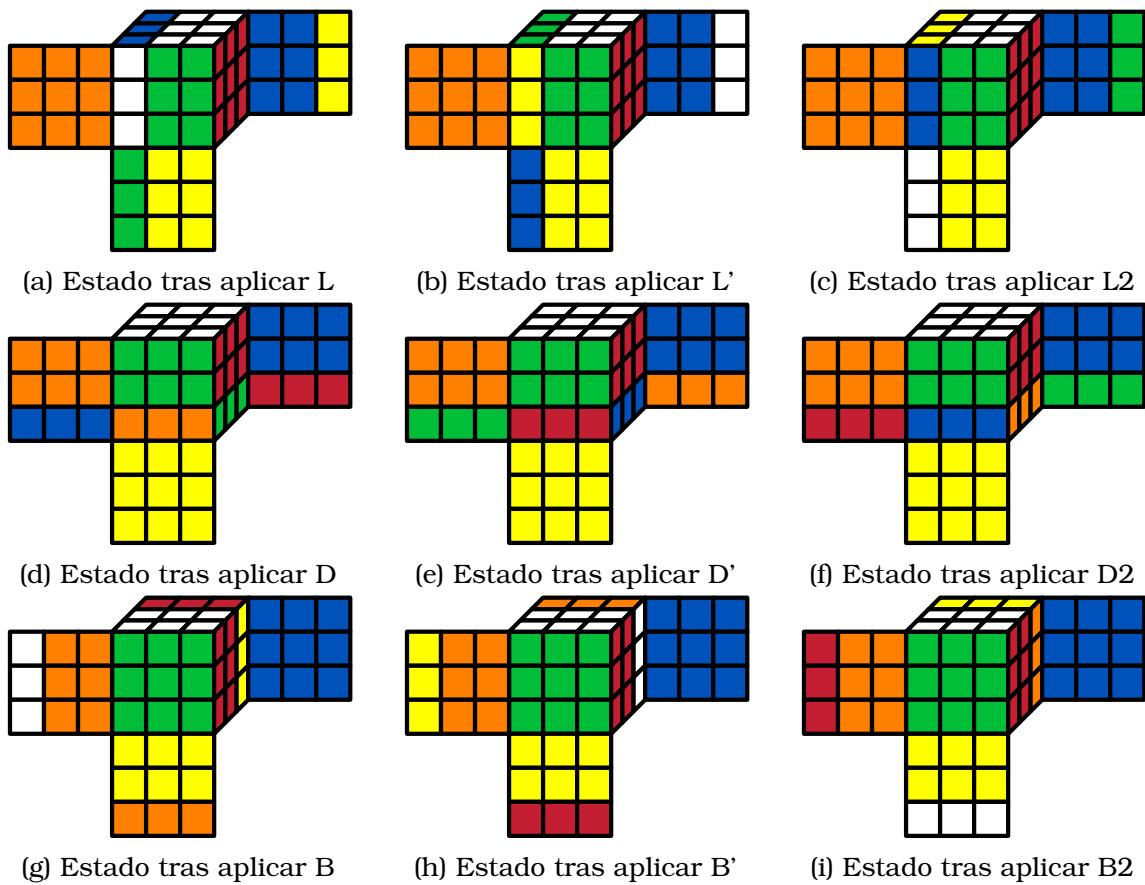


Figura 2.4: Estados tras realizar giros de las capas L, D, B

Inicial	Capa	Rotación o capa intermedia	Capa asociada
R	Right	x	R
U	Up	y	U
F	Front	z	F
L	Left	M	L
D	Down	S	F
B	Back	E	D

Cuadro 2.2: Resumen de la notación para cubos de Rubik

Capítulo 2. Preliminares

Para poner la notación en práctica, se muestra un ejemplo de la ejecución de un algoritmo. Como desambiguación, en el campo de los cubos de Rubik, se utiliza la palabra *algoritmo* como una secuencia de giros que realizan una permutación determinada, lo cual tiene un significado muy diferente con el frecuentemente utilizado en el campo de la informática. En este trabajo, se tratará de utilizar *secuencia de movimientos* como desambiguación, aunque se ha de sobreentender cuándo la palabra *algoritmo* se utiliza con su significado usual en informática y cuándo con el significado de los puzzles de permutaciones. A continuación se muestra la permutación efectuada por un algoritmo. Por convenio, al inicio el cubo suele colocarse con la cara verde como frontal y la cara blanca como cara superior. El algoritmo es **R U R' U' R' F R R U' R' U' R U R' F'** (denominada permutación tipo T). Existe una biblioteca en *LATEX*[15] que permite representar algoritmos con una notación gráfica, más idónea para quienes no conocen la notación de Singmaster. Cuando sea posible, se representarán las secuencias de movimientos con esta librería, a sabiendas de que cuenta con limitaciones, como por ejemplo no poder indicar los giros dobles. El siguiente algoritmo transforma el estado resuelto en la posición 2.5b.

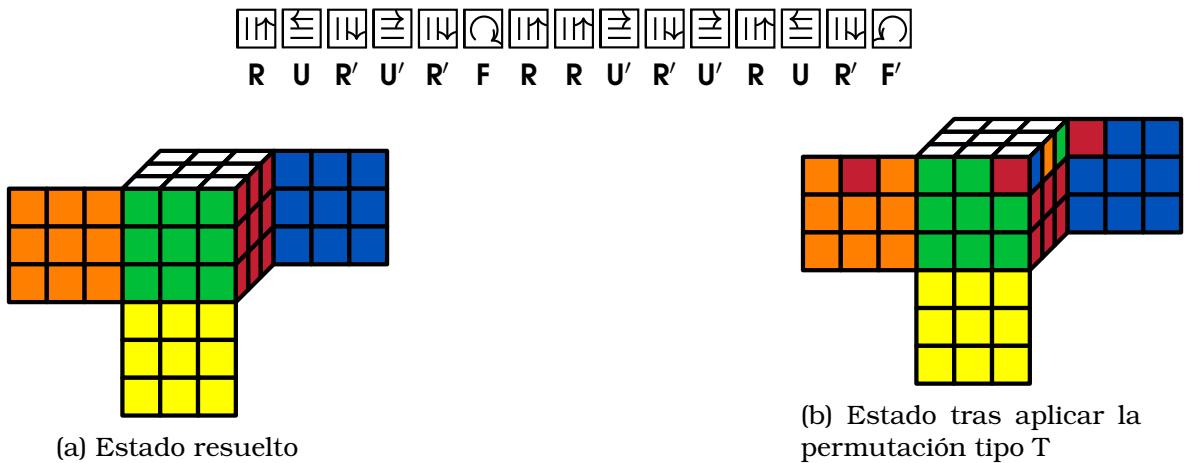
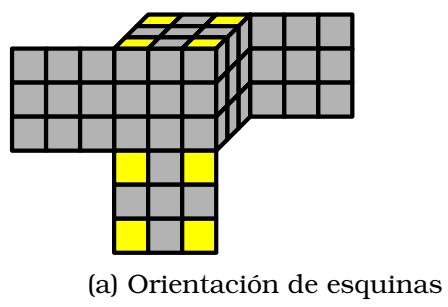


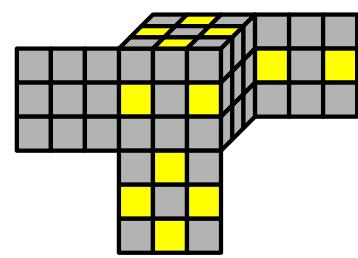
Figura 2.5: Permutación tipo T

2.1.3. Orientación correcta de las piezas

Decimos que una pieza está orientada correctamente si puede ser resuelta moviendo solamente las capas **R, U, L, D**. A nivel visual, podemos identificarlo cuando las pegatinas marcadas en la figura 2.6 están en una posición también marcada.



(a) Orientación de esquinas



(b) Orientación de aristas

Figura 2.6: Orientación de piezas

2.2. Cubos con bloqueos

Un cubo con bloqueos resulta de unir varias piezas en un cubo. De esta manera, además de tener cubos de piezas $1 \times 1 \times 1$ (de tipos aristas, esquinas y centros), podremos tener bloques de paralelepípedos de mayores dimensiones. Por poner algunos ejemplos, podrán aparecer bloques $2 \times 1 \times 1$, de tipos arista-esquina o centro-arista, o también puede aparecer un bloque $3 \times 1 \times 1$, sea que una 2 aristas con un centro o 2 esquinas, con una arista. Nada nos impide crear bloques con demasiadas piezas, por ejemplo que bloqueen una cara entera ($3 \times 3 \times 1$), o por qué no, que terminen bloqueando todo el cubo. Estos cubos serán muy sencillos o incluso triviales, dado que tendrán muy pocos estados posibles. Dependiendo de las características de un bloque, cada uno de ellos puede comportarse de forma diferente: si un bloque une a 2 o más centros, se mantendrá inmóvil, dado que ninguna de esas caras podrá girar. Si cuenta con un único centro, podrá ser rotado en 4 posibles orientaciones, pero no tendrá más movilidad que esa. Sin embargo, si solamente une esquinas y aristas, y no incluye ningún centro, el bloque podrá (en principio) moverse libremente alrededor del cubo. Evidentemente, pueden aparecer varios tipos de bloqueos en el mismo cubo. Como podemos observar, las posibilidades son muy altas. Además, dependiendo de la cantidad de bloques, sus posiciones y relación entre ellos, pueden provocar que algunas situaciones que a priori eran accesibles desde el estado resuelto dejen de serlo. Los bloques pueden restringir algunas de las propiedades de un cubo estándar. Normalmente, lo que hace complejo a un cubo de este tipo no son solamente los bloqueos con los que cuenta, sino la combinación total de ellos. Es frecuente tratar de girarlo y comenzar a bloquearse continuamente. De hecho, tampoco es extraño intentar mezclar algunos con muchos bloques, y a los pocos giros volver al estado resuelto. Estos cubos pueden ser complejos de deshacer de manera correcta. Podríamos pensar que al añadir más restricciones, el número máximo para resolver un cubo *bandaged* de forma óptima seguiría siendo 20 o menos. No obstante, esto no es cierto. Por ejemplo, el cubo *Bicube* tiene un número de 28 [4]. Existen pocos cálculos de este número para estos cubos.

2.3. Métodos de resolución del cubo de Rubik

Evidentemente, los métodos de resolución humanos son diferentes a los métodos computacionales. Los algoritmos computacionales son capaces de explorar árboles de búsqueda de miles de estados en cuestión de segundos. Por el contrario, para que un humano pueda recordar un método, ha de centrarse en resolver un pequeño número de piezas en cada paso, y buscar que cada paso sea lo más simple posible (en especial en los métodos más básicos). A nivel de longitud de la solución encontrada, esto es muy ineficiente comparado con los métodos ejecutados por máquinas. Ambos tipos pueden contar con variantes.

2.3.1. Métodos humanos

A pesar de que existen multitud de métodos de resolución [16], se van a explicar los más importantes e influyentes. El primer método que surgió fue el método

2.3. Métodos de resolución del cubo de Rubik

Corner First, desarrollado por Ernő Rubik. Este consiste en resolver todas las esquinas en primer lugar, y posteriormente resolver (normalmente en varios pasos) las aristas. El método que suele enseñarse por primera vez es denominado *método principiantes* o *Método de capa por capa (LBL, por sus siglas en inglés)* [14]. Este consiste en resolver una por una las capas del cubo, donde alguna de las capas se puede subdividir más pasos (en especial la última). El método más utilizado en la resolución de velocidad es el método *Fridrich*, o *CFOP*, el cual está basado en el método principiantes. Este consiste en cuatro pasos: se realiza una cruz, después las 2 primeras capas simultáneamente (*First 2 Layers, F2L*), y la última capa se resuelve en 2 pasos. En primer lugar, se orientan las piezas (*Orient Last Layer, OLL*), de forma que la cara superior quede resuelta sin importar que las piezas no estén en su posición correcta. Por último, se permutan estas piezas (*Permute Last Layer, PLL*). Las siglas *CFOP* representan los pasos de la resolución (*Cross, F2L, OLL, PLL*). Las ilustraciones 2.7 y 2.8 muestran los pasos de resolución de los métodos *LBL* y *CFOP* respectivamente. En todos los casos, notamos que la estrategia de resolución suele consistir en resolver algunas piezas (o su orientación) en cada etapa.

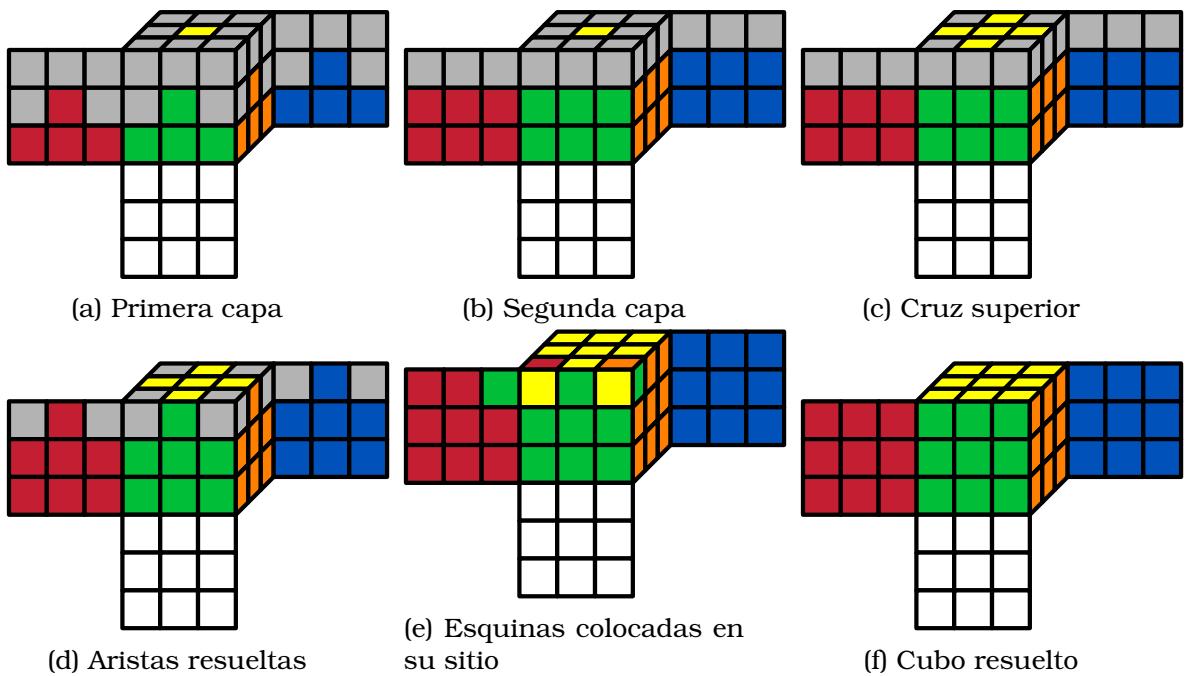


Figura 2.7: Pasos del método principiantes

2.3.2. Métodos computacionales

Los problemas de resolución de puzzles tridimensionales se reducen a problemas de búsqueda. Cuando el espacio de búsqueda de estos puzzles es relativamente pequeño, pueden resolverse mediante algoritmos de búsqueda ciega, como búsqueda en anchura con un gasto de memoria y tiempo razonable. Por ejemplo, cuando los espacios son del orden de unos pocos millones, hay software capaz de recorrerlo en tiempos menores a un minuto sin excesivas optimizaciones de

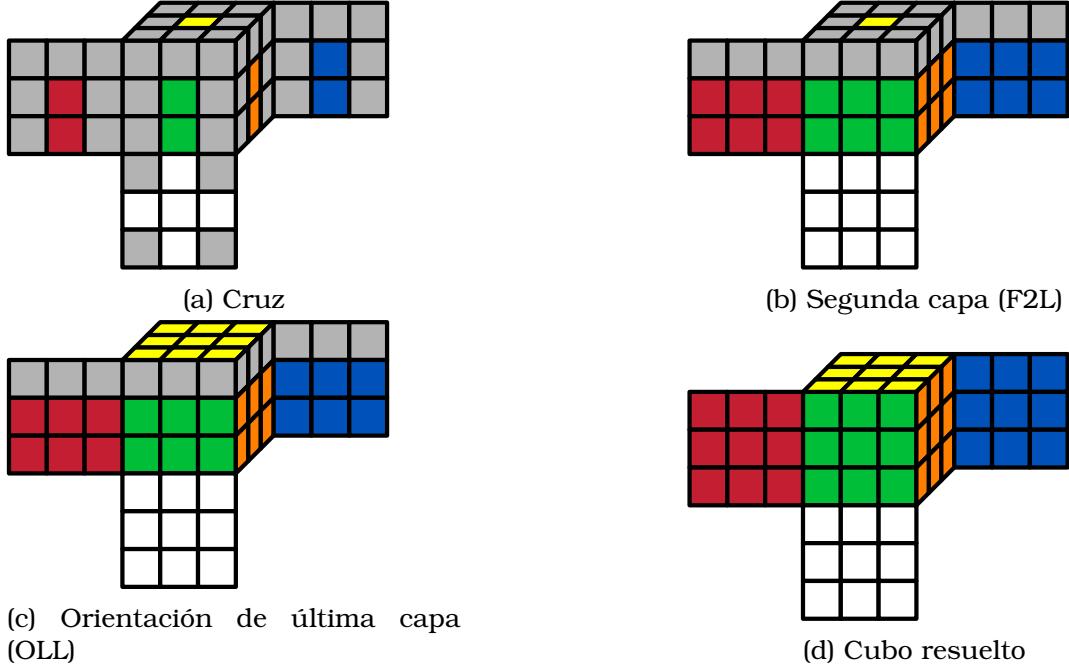


Figura 2.8: Pasos del método CFOP

memoria (existen programas para el 2x2, skewb y pyraminx en mi propio GitHub [17]). No obstante, para espacios de búsqueda más extensos, este problema se convierte en intratable. No es viable recorrer todos los estados del espacio de búsqueda del cubo de Rubik, puesto que contiene aproximadamente 43 trillones de estados diferentes (ver sección 2.4). Para ello, es necesario buscar otras alternativas para optimizar las búsquedas. Por ejemplo, pueden utilizarse algoritmos heurísticos con buena eficiencia en memoria, como el algoritmo IDA*. También se pueden emplear bases de datos de patrones [11] para obtener las soluciones o generar heurísticas. Un ejemplo de algoritmo que utiliza estos métodos es el algoritmo de *Korf*, el cual emplea 3 bases de datos de patrones a modo de heurística. Este algoritmo encuentra soluciones óptimas, aunque puede ser demasiado lento con estados lejanos al resuelto. Otra alternativa es dividir las búsquedas en varias etapas, por ejemplo resolviendo unas piezas concretas en cada una. Aunque esto podría ser explorado, no suele ser el método más utilizado. Normalmente, en cada etapa se busca acceder a permutaciones de subgrupos del cubo cada vez más restrictivos. Algunos ejemplos son el algoritmo de *Thistlethwaite* [18] o el algoritmo de *Kociemba* [19], los cuales no garantizan que la solución hallada sea óptima.

2.3.3. Cubos con bloqueos

Para resolver estos cubos, una alternativa consiste en tratar de adaptar los métodos ya existentes para cubos sin bloqueos. Esto puede ser posible para cubos con pocas restricciones, pero esto suele ser imposible para cubos con bastantes bloqueos. Por ello, normalmente se aborda su resolución como un cubo completamente nuevo, puesto que no se pueden utilizar los métodos y técnicas

2.3. Métodos de resolución del cubo de Rubik

conocidos de un 3x3 convencional. Se suele empezar estudiando las posibles ubicaciones de los bloques y las formas de acceder de unas a otras. Es de gran utilidad tener alguna visualización en forma de grafo de estos estados y sus interconexiones. De esta manera, se podrá acceder desde cualquier estado a uno indistinguible del resuelto a nivel de bloques. El siguiente paso será intercambiar las piezas y bloques que sean del mismo tipo. Esto se consigue buscando secuencias que intercambien u orienten pocas piezas y no modifiquen la geometría de los bloques. Para ello, se suele buscar un ciclo en el grafo que pase por varias geometrías y vuelva a la de partida, y analizar el resultado que ha efectuado. Con las secuencias necesarias, se podrán combinar entre ellas con un orden determinado para poder resolver las piezas, y por ello el cubo. A nivel computacional, no existe ningún solucionador general con un buen desempeño.

2.4. Introducción a la teoría de grupos

2.4.1. Definición

Un grupo es una estructura algebraica. Dado un conjunto G y una operación binaria (\cdot) , decimos que (G, \cdot) es un grupo cuando cumple las siguientes propiedades:

- La operación es interna: $(\cdot) : G \times G \rightarrow G$,
 $\forall x, y \in G, (x \cdot y) \in G$
- La operación es asociativa: $\forall x, y \in G, (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Existe un elemento neutro: $\exists e \in G / \forall x, xe = ex = x$
- Todo elemento tiene un inverso: $\forall x \in G, \exists (x^{-1}) \in G / x \cdot x^{-1} = x^{-1} \cdot x = e$

2.4.2. Grupos de permutaciones

S_n representa el grupo de permutaciones de n elementos, donde hay $n!$ opciones. Este conjunto con la operación de componer permutaciones forma un grupo.

Normalmente los $\sigma \in S_n$ se representan de la siguiente manera: $\begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix}$

Aunque la primera fila puede obviarse y representar cada elemento mediante una lista en orden, es decir, la segunda fila.

Por ejemplo, los elementos de S_3 serían

[1, 2, 3]

[1, 3, 2]

[2, 1, 3]

[2, 3, 1]

[3, 1, 2]

[3, 2, 1]

Se denomina *soporte* a los elementos que son permutados tras aplicar una permutación.

$$sop(\sigma) = \{i \in \{1, \dots, n\} | \sigma(i) \neq i\}$$

Por el contrario, llamaremos *puntos fijos de una permutación* (abreviado medianamente *pf*) a aquellos elementos que se mantienen fijos tras aplicar una permutación, es decir:

$$pf(\sigma) = \{i \in \{1, \dots, n\} | \sigma(i) = i\}$$

Veremos que esto será útil para definir ciertos cálculos sobre las piezas de un cubo de Rubik.

2.4.3. Grupo del cubo de Rubik

Un cubo de Rubik es un grupo no conmutativo, donde G es el conjunto de todas las permutaciones posibles. Podemos entender que un estado de un cubo es equivalente a una permutación concreta de piezas. La operación binaria sería concatenar las 2 permutaciones una tras otra. La operación es interna, puesto que tras concatenar 2 permutaciones encontramos una permutación nueva del cubo. Es asociativa, puesto que si se realizan varias secuencias de giros, se alcanza el mismo estado sin importar cómo se asocien. El elemento neutro sería el estado resuelto, es decir no intercambiar ninguna pieza ni efectuar ningún giro. Existe un inverso, el cual sería deshacer los giros que permutan una serie de piezas. El número de estados posibles en un cubo de Rubik es de aproximadamente 43 trillones. Aunque este número ha sido calculado en multitud de fuentes, se encuentra su cálculo detallado en la página 12 del libro [14]. Corresponde a:

$$|G| = \frac{8! \cdot 3^8 \cdot 12! \cdot 2^{12}}{12}$$

Matemáticamente, se suele representar G como:

$$G \approx S_8 \times \mathbb{Z}_3^8 \times S_{12} \times \mathbb{Z}_2^{12}$$

\mathbb{Z}_p^n representa n números del conjunto $\{0, 1, \dots, p-1\}$. Por ejemplo, un posible elemento de \mathbb{Z}_3^8 sería $(0, 1, 2, 0, 1, 2, 0, 0)$.

De esta manera, los grupos de permutaciones S_n hacen referencia a la forma de intercambiar las piezas, y los conjuntos de \mathbb{Z}_p^n a las maneras de orientar cada pieza. Aquellos con $n = 8$ hacen referencia a las esquinas, mientras que aquellos con $n = 12$ representan las aristas. Sería necesario realizar una numeración previa de cada pieza para poder codificar esta información. Cabe destacar que los centros son fijos, y por ello no los tenemos en cuenta, aunque dependiendo de la codificación a usar podrá ser o no útil almacenarlos.

2.4.4. Grupoide de los cubos con bloqueos

Pueden existir cubos con bloqueos que sean subgrupos de un cubo 3x3x3. Por ejemplo, puede ocurrir cuando restringen por completo algunas capas y permiten giros con total libertad en otras (únicamente colocando bloques de varios centros). También puede ocurrir si fuerzan a algunas capas a ser giradas con movimientos de 180 grados (por ejemplo, con bloques 3x1x1 que unan un centro y 2 aristas). No obstante, esto no suele ser el caso. Normalmente, los cubos con bloqueos no son grupos, sino grupoideos. Esto se debe a que no siempre podemos componer una operación para 2 elementos (o con una visualización en el cubo, no siempre podremos girar una capa, debido a que esté bloqueada en una configuración). Donde sí que podamos, se verificarán las propiedades de grupo (asociatividad, elemento neutro e inverso). Desafortunadamente, esta estructura no nos aporta propiedades de grupos que podamos aprovechar de manera general.

2.5. Introducción a la programación funcional y Haskell

La programación funcional es un paradigma de programación que está basado en el cálculo lambda y en las funciones puras. Estas son aquellas que no tienen efectos secundarios y que siempre devuelven el mismo resultado para cierto input. Para permitir esto, los lenguajes funcionales evitan al máximo las estructuras de datos mutables y la computación basada en estados o en iteración. Por contra, utilizan datos inmutables (aquellos que no modifican su estado una vez creados, y que para ser modificados es necesario crear una copia), y la computación se basa en recursividad. Esto aporta una mayor seguridad y ausencia de *side effects* a la hora de programar, y en ocasiones permite un código muy conciso y elegante. Las funciones son ciudadanos de primera clase, es decir, pueden ser pasadas como argumento a una función, ser devueltas por otra función o estar contenidas en una estructura de datos. Esto permite un código altamente reutilizable. Además, estos lenguajes suelen tener gran influencia del lenguaje matemático, por lo que es idóneo para este tipo de proyectos. Aunque los lenguajes funcionales puros no son los más populares, han influido en otros lenguajes como Java, Python o Javascript, y cada vez es más común encontrarse con características funcionales en lenguajes imperativos u orientados a objetos. La mayoría de este proyecto se realiza en *Haskell*. Este es un lenguaje funcional puro², que cuenta con tipado estático, inferencia de tipos y evaluación perezosa. También cuenta con curryficación y aplicación parcial. Esto significa que una función a la que se le aplican menos argumentos de los requeridos devuelve una nueva función (con los primeros valores evaluados). *Haskell* permite definir tipos algebraicos y clases de tipos. Esto puede ser útil para definir conceptos más similares a la forma humana de pensar, lo que puede ser interesante para realizar un código claro y sencillo de mantener.

Ejemplos de código en Haskell

```
--Función que eleva un entero al cuadrado
square :: Int -> Int
square x = x * x

--Cálculo del factorial de un entero
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)

--Cálculo de las ternas pitagóricas
menores que 10 (mediante listas por comprensión)
pitagoric :: Int -> [(Int, Int, Int)]
pitagoric n = [(x,y,z) |
  x <- xs, y <- xs, z <- xs,
  x^2 + y^2 == z^2, x <= y]
  where xs = [0..n]
```

²Solamente soporta características funcionales. Existen lenguajes multi-paradigma, que soportan funcionalidades de varios paradigmas.

2.5. Introducción a la programación funcional y Haskell

```
--Función que duplica todos los números de una lista
duplicateList :: [Int] -> [Int]
duplicateList = map (*2)
--Sin curryficación, equivale a:
--duplicateList xs = map (*2) xs

--Descartar los números impares
y a los pares aplicarles una función
fEven :: [Int] -> (Int -> Int) -> [Int]
fEven xs f = (map f . filter even) xs
```


Capítulo 3

Análisis y diseño

3.1. Requisitos

A nivel de interacción con el usuario, el software tendrá los siguientes requisitos funcionales:

- Introducir un estado arbitrario de un cubo de Rubik.
- Introducir unos bloqueos determinados de un *bandaged cube*.
- Procesar el input y detectar posibles errores en él.
- Generar una visualización del estado introducido para verificar el input.
- Encontrar una secuencia de movimientos que resuelva el cubo.
- Generar una visualización que realice los giros pertinentes y termine con el cubo resuelto.

Los requisitos no funcionales son los siguientes:

- El programa debe de ser confiable. Siempre que encuentre una solución del cubo, será correcta y resolverá el cubo. Siempre que exista una solución, el software ha de encontrarla.
- El motor de búsqueda será lo más rápido y eficiente posible. Debe de hallar la solución sin agotar los recursos de la máquina. El tiempo en encontrar una solución ha de ser razonable y lo menor posible.
- El software debe de ser tan usable y sencillo de utilizar como sea posible.

Estas funcionalidades son las que utiliza un usuario en una iteración estándar. No obstante, el software tiene muchas más funcionalidades a nivel interno. Esto aporta una biblioteca de funciones que puede ser utilizada en otros contextos, lo que permite realizar cálculos que no solo sean soluciones a un cubo con bloqueos. Además, facilitaría la integración de expansiones en un futuro.

Otras funcionalidades con las que cuenta la aplicación a nivel interno son:

Capítulo 3. Análisis y diseño

- Procesar el input y completar los bloques que no se introduzcan completamente. Calcular el menor paralelepípedo que englobe a varias piezas.
- Representar y modelar un cubo de Rubik con o sin bloqueos. Convertir su definición a varias representaciones diferentes.
- Calcular los estados a los que accede un cubo tras aplicar una secuencia de movimientos.
- Comprobar si una secuencia de movimientos preserva o rompe los bloques de un cubo.
- Generar una visualización de cualquier estado de un cubo y aplicar sobre él cualquier secuencia de movimientos.
- Generar y utilizar las heurísticas de Korf para un cubo. Esto aportará una cota inferior del número de movimientos necesarios para resolverlo.
- Implementar el algoritmo de Korf para resolver un cubo con o sin bloqueos.

3.2. Casos de uso

A nivel de interacción con el usuario, la aplicación es bastante simple. En primer lugar, el usuario introducirá el input como descripción de un estado de un cubo. El programa solicitará los colores de las piezas en un orden determinado, y posteriormente los bloques que restringen el cubo. Si el programa detecta algún error en el input, solicitará al usuario que vuelva a introducir los datos correspondientes. Una vez tenga la información necesaria, devolverá la solución que resuelve el cubo, además de una visualización de los movimientos que lo resuelve. Se muestran estos pasos en un diagrama de secuencia en la figura 3.1.

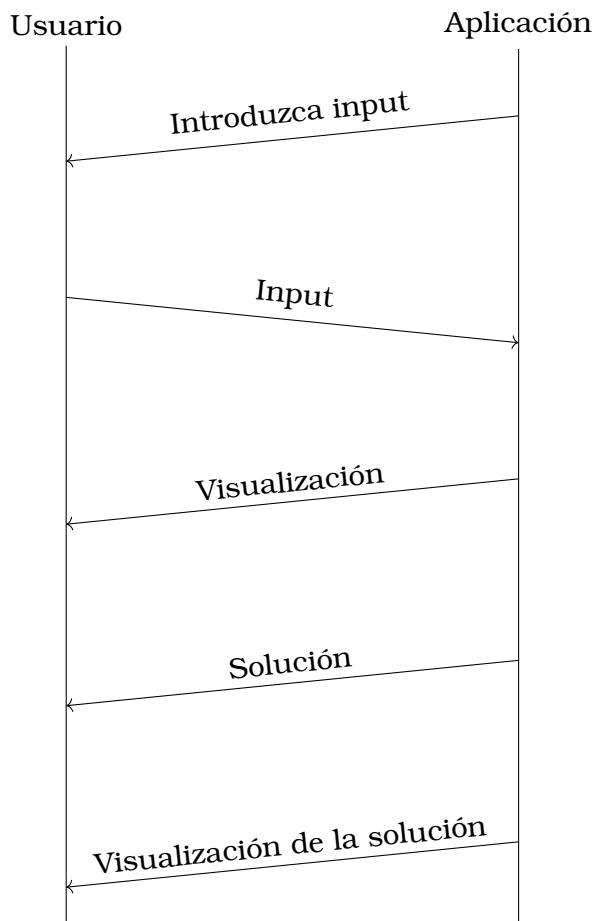


Figura 3.1: Diagrama de secuencia de la interacción del usuario con el sistema

3.3. Estructura y división en módulos

Para diseñar las partes necesarias del programa, podremos dividirlo en 4 paquetes. Cada uno de ellos tendrá sus correspondientes módulos. El paquete *Data* contendrá lo necesario para codificar toda la información de los cubos. El paquete *IO-Visualizator* servirá para gestionar el input y output de un cubo, además de

Capítulo 3. Análisis y diseño

mostrar una visualización por pantalla. El paquete *Heuristics* gestionará la generación, indexado y acceso a las bases de datos de las heurísticas de Korf. Este último será utilizado por el paquete *Search-Engine*, que encontrará una solución a un cubo arbitrario. A continuación, se muestra un diagrama de los paquetes del programa y sus dependencias en la figura 3.2. En estos se indica con una flecha discontinua una dependencia débil, y mediante una flecha continua el uso de algunas funciones o datos.

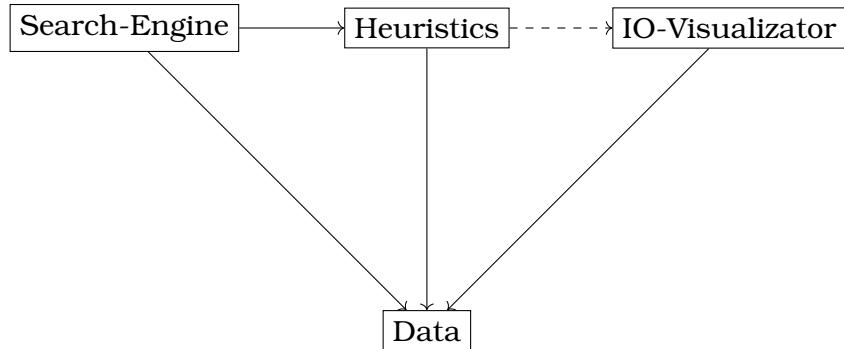


Figure 3.2: Diagrama de paquetes y dependencias

En cada paso, el usuario aportará un input al programa y le devolverá una solución. Internamente, el paquete *IO-Visualizer* solicitará un input al usuario, y utilizará las funciones de *Data* para procesar el input y calcular una representación de un cubo. Posteriormente, se llamará al paquete *Search-Engine*, que buscará una solución con ayuda del paquete *Heuristics*. Esta solución se devolverá a *IO-Visualizer*, el cual devolverá al usuario una visualización por pantalla de los movimientos que resuelven el cubo. El diagrama de lo que calcula cada módulo se muestra en la figura 3.3.

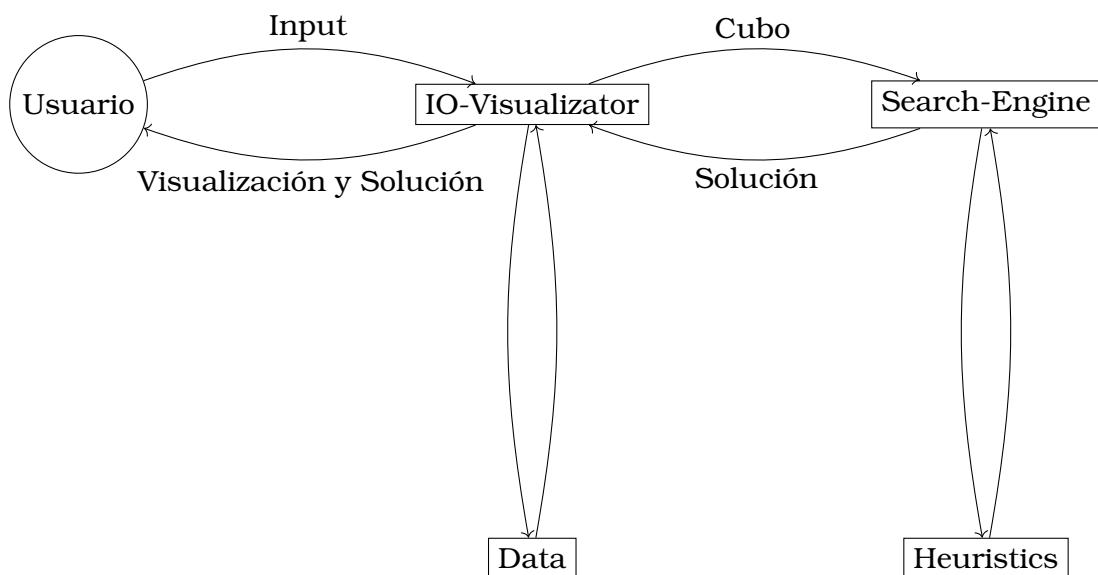


Figure 3.3: Diagrama de la información de los paquetes

3.3. Estructura y división en módulos

Una vez hemos diseñado los paquetes, se puede dividir cada paquete en sus correspondientes módulos. El paquete de datos se puede diseñar con 4 módulos: *Mathematical-Notation*, *Bandaged*, *Cube* y *Moves*. El módulo *Cube* implementa la representación de un cubo estándar. El paquete *Moves* define los movimientos posibles y su efecto en los cubos. El módulo *Bandaged* permite representar un cubo con bloqueos. Por último, *Mathematical-Notation* permite convertir un cubo en su notación matemática, como elemento de $S_8 \times \mathbb{Z}_3^8 \times S_{12} \times \mathbb{Z}_2^{12}$. El diagrama de dependencias se muestra en la figura 3.4.

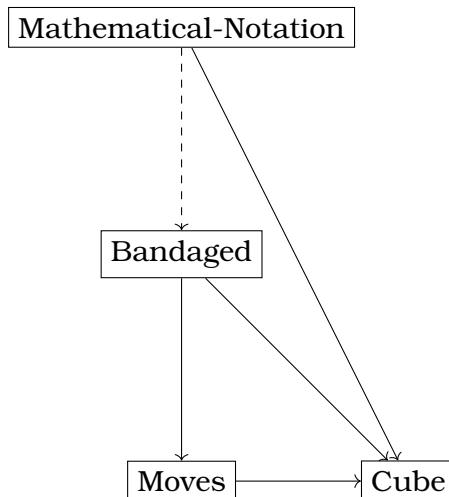


Figure 3.4: Módulos del paquete *Data*

El paquete *IO-Visualizator* contiene 5 módulos. *InputCube* es el más importante, puesto que es llamado desde otros paquetes. Este utiliza *Visualizator* para mostrar por pantalla un cubo y sus movimientos para resolverse. *ManimHsConversion* permitirá convertir los datos de *Haskell* con los requeridos por la librería de *Manim*. También utilizará funciones de *InputBandagedCube*, que permite devolver un cubo con bloqueos de manera relativamente sencilla. Este último utilizará funciones de *ExpandBlocks*, que permiten realizar la compleción de bloques. El diagrama de dependencias se muestra en la figura 3.5.

Capítulo 3. Análisis y diseño

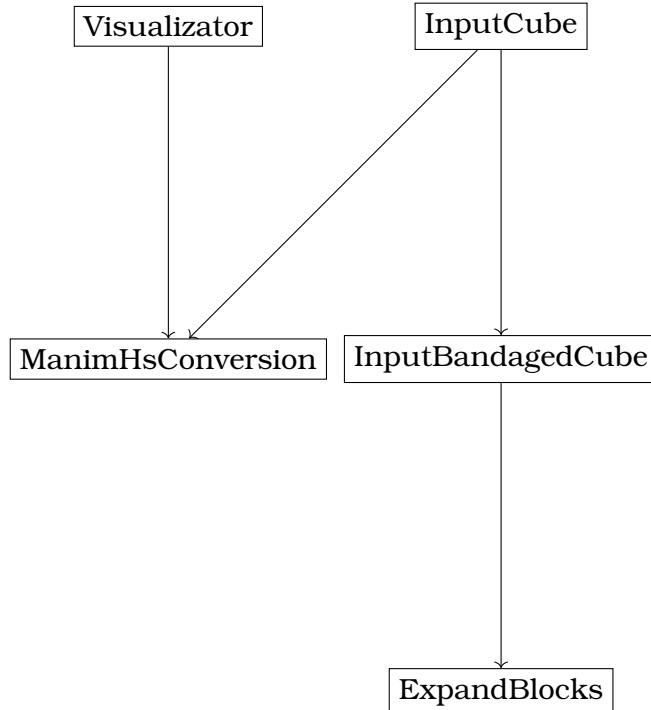


Figure 3.5: Módulos del paquete *IO-Visualizer*

El paquete *Search-Engine* cuenta con 3 módulos. *SolvingStrategies* es el más importante, dado que sus funciones son llamadas para resolver un cubo desde el main. Este gestiona las llamadas a los módulos del paquete, calculando y pasando los argumentos necesarios. Cuenta con varias estrategias (iddfs, algoritmo de Kociemba y algoritmo de Korf), pero la que mejor funciona para los cubos con bloqueos es sin duda el algoritmo de Korf. El paquete *Search* implementa las búsquedas mediante el algoritmo IDA*, y recomponen la solución en caso de haberla. Este contará con una búsqueda genérica que tome el predicado a cumplir, la heurística a utilizar y las capas que pueden moverse en un cubo. Por último, el módulo *MoveGeneration* facilita el cálculo de las capas no bloqueadas y los movimientos que pueden ser pasados al algoritmo genérico. El gráfico de dependencias de este paquete se muestra en la figura 3.6.

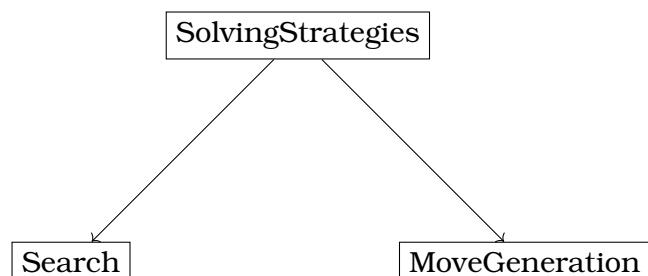


Figure 3.6: Módulos del paquete *Search-Engine*

Por último, el paquete *Heuristics* se divide en 4 módulos. El más importante es

3.4. Notación de entrada y definición de bloques

KorfHeuristic, cuyas funciones son llamadas desde otros paquetes. Este permite utilizar las heurísticas de Korf de forma cómoda, y se encargará del resto de llamadas a funciones de otros módulos. Este utilizará el módulo *GenKorfHeuristic*, que generará las heurísticas por primera vez. Se almacenarán en un vector con una indexación estratégica, calculada por el módulo *IndexHeuristic*. Este a su vez requerirá cálculos combinatorios, aportados por el módulo *Combinatorics*. El diagrama de dependencias se muestra en la figura 3.7.

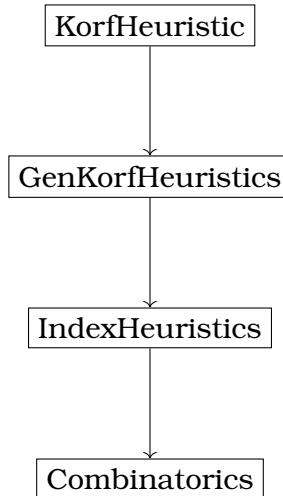


Figure 3.7: Módulos del paquete *KorfHeuristic*

3.4. Notación de entrada y definición de bloques

Para que el usuario introduzca una posición de un cubo (estándar) al programa, ha de especificar de qué color es cada pieza. Para ello, el programa le dará un orden de caras, y en cada cara los introducirá por filas (ver figura 3.8). Posteriormente, el programa se encargará de revisar si la posición es correcta y de convertirlo en una representación más eficiente a nivel computacional (ver figura 3.9). Para introducir los bloqueos, se solicitará que introduzca las piezas bloqueadas (Ejemplo: “Rojo + Rojo-Verde + Rojo-Blanco + Rojo-Blanco-Verde, Azul + Azul-Amarillo”). El programa puede adivinar qué tipo de pieza es cada una en función del número de pegatinas que contiene. Tras esto, el programa revisará que el input puede ser correcto. Es posible que el usuario no introduzca todas las piezas de cada bloque, por lo que el programa ha de calcular, para cada bloque, el menor paralelepípedo que englobe a todas las piezas introducidas. A este paso lo llamaremos “compleción de bloques”.

3.4.1. Compleción de bloques

Al recibir el input, lo primero que debe de hacer el código es procesarlo. Según se especifiquen ciertos bloqueos, es posible que el usuario no haya introducido todas las pegatinas que lo forman, o que sea imposible separar algunas piezas de este bloque. Por tanto, el programa realiza una “compleción de bloques”, para

Capítulo 3. Análisis y diseño

	1	2	3									
	4	5	6									
	7	8	9									
37	38	39	19	20	21	10	11	12	46	47	48	
40	41	42	22	23	24	13	14	15	49	50	51	
43	44	45	25	26	27	16	17	18	52	53	54	
			28	29	30							
			31	32	33							
			34	35	36							

Figura 3.8: Orden en el que introducir los colores de cada pieza

encontrar el mayor paralelepípedo que incluya a todas las pegatinas especificadas en cada bloque. También se unirán los bloques con piezas en común, de forma que el programa devuelva la menor cantidad de bloqueos necesarios para cada caso. Este procedimiento se realiza desde el estado resuelto, aunque también sería válido calcularla desde la posición especificada por el usuario. Supongamos que se han introducido una serie de pegatinas en cada bloque. El primer paso para procesarlo será realizar una unión disjunta de bloques. Cuando haya varios bloques con algún elemento en común, se realizará su unión. Este proceso se puede repetir por transitividad (no profundizaremos en esto, pero la relación de pertenecer al mismo bloque es una relación de equivalencia). Más adelante, se debe de calcular para cada bloque el mayor paralelepípedo que contenga todas sus piezas. Para ello, podemos realizar un procedimiento similar al de revisar si un giro preserva un bloque. Una vez sabemos que un giro divide el cubo en soporte y puntos fijos, y que un giro preserva un bloque cuando el bloque es disjunto a uno de ellos, sabemos que el único subconjunto con elementos en común con el bloque contiene a todas las piezas que ese giro no puede separar del bloque (para un giro que no rompe el bloque). El mayor paralelepípedo viene dado por la intersección de todas estas piezas inseparables del bloque de cada giro, siempre y cuando preserven el bloque. Puede ser tentador pensar que solo habría que utilizar los movimientos válidos, pero no nos ha de importar que rompan otros bloques. Al realizarse con un giro de cada capa, estaríamos calculándolo para todas las posiciones y orientaciones del bloque tras cualquier número de giros. Más formalmente, su cálculo se detalla a continuación. Sean:

- El conjunto de piezas: $P = \{0, 1, \dots, 53\}$

3.4. Notación de entrada y definición de bloques

- b_i un bloque: $b_i \subset \wp(P)$
- B el conjunto de bloques del cubo: $B = \{b_1, b_2, \dots, b_n\}$
- σ un giro: $\sigma \in \{R, R', R2, U, \dots, \}$
- Los puntos fijos del giro, es decir, los que no se intercambian:
 $pf(\sigma) = \{x \in P \mid \sigma(x) = x\}$
- El soporte del giro, es decir, los que si que la cambian:
 $sop(\sigma) = \{x \in P \mid \sigma(x) \neq x\}$

Algoritmo de compleción de bloques: En primer lugar, necesitamos unir los bloques con alguna pieza en común (por transitividad). De esta forma, obtendremos bloques disjuntos. Para ello, realizamos uniones de los conjuntos no disjuntos. Si los bloques son:

$$\text{Bloques} = \{b_1^0, b_2^0, \dots, b_m^0\}$$

con

$$b_i^0 \cap b_j^0 \neq \emptyset \text{ con algún } i \neq j$$

Unificaremos estos bloques calculando:

$$\text{Bloques-disjuntos} = \{b_1, b_2, \dots, (b_i \cup b_j), \dots, b_m\}$$

Bloques-disjuntos verifica:

$$b_n \cap b_m = \emptyset, \forall i \neq j$$

Tras esto, y a partir de Bloques-disjuntos, el mayor paralelepípedo viene dado por:

$$\text{Expansión}(b) = \bigcap_{i=1}^6 f(\sigma_i, b)$$

Donde:

$$f(\sigma_i, b) = \begin{cases} sop(\sigma_i) & \text{si } pf(\sigma_i) \cap b = \emptyset \\ pf(\sigma_i) & \text{si } sop(\sigma_i) \cap b = \emptyset \\ P & \text{en otro caso} \end{cases}$$

Realizamos la unión entre 1 y 6, uno para cada giro de una capa. Es indiferente el sentido de giro de cada una.

Ejemplo:

Supongamos que el usuario introduce como bloques las siguientes piezas: $B_0 = \{\{28, 48\}, \{30, 48\}\}$ En primer lugar, se hacen uniones de los conjuntos con alguna pieza en común. El 48 aparece en varios conjuntos. $B_1 = \{\{28, 30, 48\}\}$ Calculamos las intersecciones de cada bloque (en este caso 1) con los soportes y puntos fijos de cada giro (3.1):

El bloque expandido viene dado por:

Giro (σ)	$sop(\sigma) \cap b_i$	$pf(\sigma) \cap b_i$	$f(\sigma_i, b)$
R	{28}	{30, 48}	P
U	{28, 30, 48}	\emptyset	$sop(U)$
F	{30}	{28, 48}	P
L	\emptyset	{28, 30, 48}	$pf(L)$
D	\emptyset	{28, 30, 48}	$pf(D)$
B	\emptyset	{28, 30, 48}	$pf(B)$

Cuadro 3.1: Compleción de bloques

$$\begin{aligned}
 Exp(b) &= \bigcap_{i=1}^6 f(\sigma_i, b) \\
 &= f(R, b) \cap f(U, b) \cap f(F, b) \cap f(L, b) \cap f(D, b) \cap f(B, b) \\
 &= P \cap sop(U) \cap P \cap pf(L) \cap pf(D) \cap pf(B) \\
 &= sop(U) \cap pf(L) \cap pf(D) \cap pf(B) \\
 &= \{b_e, 0, 24, \dots\} \cap \{b_e, 26, 6, \dots\} \cap \{b_e, 32, 49, \dots\} \cap \{b_e, 53, 46, \dots\} \\
 &= b_e = \{9, 10, 11, 28, 29, 30, 31, 48\}
 \end{aligned}$$

Por cuestiones de simplicidad a la hora de escribir, se denomina a la solución final

$$b_e = \{9, 10, 11, 28, 29, 30, 31, 48\}$$

En caso de haber varios bloques no disjuntos, se realiza la misma expansión de bloques a cada uno de ellos de forma independiente.

3.5. Salida y visualización

Evidentemente, esta forma de introducir estados y bloqueos de un cubo no tiene por qué ser la más cómoda, y es bastante probable que el usuario cometa algún error. Para aportar comodidad, se va a realizar una visualización del cubo, para que el usuario pueda corroborar que el input que introdujo fue correcto. Esto se va a realizar con la herramienta *Manim*, en el lenguaje *Python*. Existe una biblioteca llamada *Manim RubiksCube* [20] que facilita esta tarea, la cual será adaptada para nuestro caso. Esta herramienta mostrará un cubo en la pantalla para que el usuario revise el input que introdujo, y rotará alrededor de una diagonal para poder visualizar las 6 caras del cubo. Si el algoritmo encuentra una solución, se generará una visualización del cubo realizando los giros pertinentes hasta resolverlo. También se imprimirá por pantalla mediante la notación habitual (ver 2.1.2), para que el usuario decida qué opción utilizar.

3.6. Modelado de los datos

Una parte fundamental de este problema reside en cómo representar y codificar la información. Esta debe de mantener un equilibrio entre la legibilidad, sencillez

de uso, el consumo de memoria y la velocidad.

3.6.1. Codificación de un cubo con y sin bloqueos

Para codificar un cubo con bloqueos como tipo de dato, podemos representar un cubo estándar, al que posteriormente le añadimos restricciones. Antes de realizar cada giro, se debe de comprobar que no rompería ningún bloque.

Codificación de un cubo de Rubik estándar La única alternativa que hemos tratado para representar numéricamente un cubo es la codificación matemática explicada en 2.4. Aunque esta codificación es muy útil y permite demostrar multitud de propiedades, no será utilizada. Esto se debe a que en cada giro se necesitan 4 reordenaciones de vectores y 20 sumas modulares. Dado que se van a realizar millones de giros en una búsqueda, conviene utilizar otras codificaciones más rápidas, aunque sean menos elegantes. No se ha probado la eficiencia experimentalmente, pero por experiencia en otros proyectos similares, esta codificación suele traer desventajas a nivel de velocidad.

Para representar un estado arbitrario de un cubo de Rubik estándar, almacenaremos las pegatinas en un vector de 54 números, cada uno representando una pegatina diferente, las cuales han de ser numeradas con anterioridad (ver figura 3.9). De esta manera, en cada giro se necesitará una reordenación de vectores. Evidentemente contiene redundancias, pero se espera que funcione con mayor velocidad. Se utilizará la biblioteca de *Haskell Data.Vector* [21], la cual permite codificar vectores similares de maneras muy eficientes. En concreto, nos interesa la función *backpermute*, puesto que permuta los elementos de un vector en complejidad lineal. Dado que cada número será menor de 63, se podrá explorar en el futuro si utilizar la base 64 (de menos de 1 Byte) en lugar de la de los enteros convencionales (de 4 Bytes). No obstante, por simplicidad y legibilidad del código, esto no se explorará hasta que la memoria de ejecución escasee. Aunque cualquier numeración de las piezas puede ser correcta y válida, podemos realizarla con algo de estrategia para simplificar algunos pasos posteriormente. La numeración a usar será la indicada en la figura 3.9:

En esta codificación, primero se numeran las esquinas, después las aristas y por último los centros. De esta manera, en caso de necesitar las posiciones de solo un tipo de pieza, utilizaremos la función *slice* de la biblioteca *Data.Vector*.

Para cada esquina, colocamos los números múltiplos de 3 en las caras superior o inferior, y el resto de números en sentido horario. Para las aristas, haremos algo similar, ubicando los números pares donde la orientación sea correcta (ver 2.1.3), y el número impar posterior ocupará la posición restante. De esta forma, podremos conocer si una pieza está orientada realizando el módulo 2 ó 3 (en complejidad constante). Por último, se numeran los centros sin condiciones adicionales. Aunque pudiéramos pensar que los centros son fijos y no merece la pena almacenarlos, existe una ventaja de conservarlos. Como veremos, algunos bloqueos pueden impedir el giro de algunas caras, dependiendo de si contienen varios centros. De esta manera, podremos reducir el factor de ramificación del árbol de búsqueda, y no generar estados que siempre van a ser inaccesibles

Capítulo 3. Análisis y diseño

	3	26	6									
	24	48	28									
	0	30	9									
4	25	2	1	31	11	10	29	8	7	27	5	
39	52	33	32	49	34	35	50	37	36	51	38	
20	47	22	23	41	13	14	43	16	17	45	19	
			21	40	12							
			46	53	42							
			18	44	15							

Figura 3.9: Codificación numérica de un estado arbitrario

(mejorando la eficiencia del algoritmo). En el vector, cada pieza será un número, y cada posición del cubo será una posición de memoria del vector. Por ejemplo, en la orientación estándar, la pegatina blanca de la pieza blanco-verde-naranja será el número 0. Si dicha pieza estuviera entre las caras amarilla, roja y azul y con la pegatina blanca en la cara amarilla, en el vector tendríamos que $v[15] = 0$. Al realizar un giro, se permutarán los números de posición dentro del vector. Un ejemplo de la posición 2.5b (permutación tipo T) en esta codificación será (se han permutado entre sí las esquinas 9-10-11 y 6-7-8 y las aristas 28-29 y 24-25):

[0, 1, 2, 3, 4, 5, 9, 10, 11, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
28, 29, 26, 27, 24, 25, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53]

Codificación de un cubo con bloqueos Una vez tenemos una numeración de piezas, cada bloqueo será un conjunto de piezas determinadas. Por ello, codificaremos un cubo con bloqueos mediante una numeración de un cubo estándar y varios conjuntos de piezas bloqueadas, es decir, un conjunto de conjuntos de números. Existe una librería en *Haskell* que permite codificar esta estructura de datos de forma muy eficiente (Data.Set, [22]). Se basa en árboles binarios balanceados en altura, y muchas de sus operaciones se realizan en complejidad logarítmica con respecto al número de elementos.

Ejemplo: Estado resuelto con los bloqueos: “Rojo + Rojo-Verde + Rojo-Blanco +

Rojo-Blanco-Verde, Azul + Azul-Amarillo".

Posición = $[0, 1, \dots, 53]$

Bloqueos = $\{\{9, 10, 11, 34, 35, 28, 29, 50\}, \{44, 45, 51\}\}$

3.6.2. Codificación de giros

Cubos sin bloqueos Existen varias maneras de representar un giro sobre un cubo. Una alternativa, y posiblemente la más intuitiva a nivel humano es describiendo el cubo como una acción de un grupo sobre un conjunto: de esta manera, tenemos el conjunto formado por todos los estados posibles del cubo (denominado como X), y tenemos el conjunto de giros ($G = \{R, R', R2, U, U', U2, \dots\}$). Cada giro es una función que toma un estado de X y devuelve otro de X , y ocurre lo mismo para cualquier composición de giros. Otra alternativa (la que se va a utilizar), será definir una operación entre 2 estados arbitrarios de X , donde se reordenan las piezas del primer cubo con el orden del segundo. Necesitamos codificar la permutación que efectúa cada giro, para después poder componerlas. A la hora de aplicar un giro a un estado, lo que realizaremos será componer el estado inicial con el reordenamiento que realiza un giro (ver ejemplo más adelante). Será necesario almacenar los reordamientos de los 18 giros posibles. Evidentemente, para ejecutar varios giros tendremos que componer esta función varias veces para realizar todos los giros uno por uno. Vamos a utilizar esta última codificación, puesto que se adapta algo mejor a las funciones que aportan las bibliotecas *Data.Group* y *Data.Vector*. No obstante, no se descarta probar ambas en el futuro y comparar sus rendimientos o legibilidad de cada implementación.

Ejemplo de aplicar un giro: Sea x el estado inicial (permutación tipo T).

$$x = [0, 1, 2, 3, 4, 5, 9, 10, 11, 6, 7, 8, 12, 13, \dots, 22, 23, 28, 29, 26, 27, 24, 25, 30, \dots, 52, 53]$$

Y la permutación que efectúa el movimiento U, a la cual llamaremos y (es necesario almacenar las permutaciones del resto de movimientos):

$$y = permOfTurnU = [9, 10, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, \dots, 22, 23, 30, 31, 24, 25, 26, 27, 28, 29, 32, 33, 34, \dots, 53]$$

Aplicar el movimiento U a x será:

$$\begin{aligned} U(x) = & x \cdot permOfTurn(U) = x \cdot y = \\ & x[y[0]], x[y[1]], \dots, x[y[53]] = \\ & x[9], x[10], x[11], x[0], \dots, x[53] = \\ & [6, 7, 8, 0, 1, 2, 3, 4, 5, 9, 10, 11, 12, 13, \dots, \\ & 23, 30, 31, 28, 29, 26, 27, 24, 25, 32, 33, \dots, 53] \end{aligned}$$

Este procedimiento es válido para cualquier cubo estándar. Cuando haya bloqueos, será necesario comprobar antes que dicho giro no rompe ningún bloque.

Capítulo 3. Análisis y diseño

Cubo con bloques Para verificar la validez de un giro, podemos entender que cada uno divide el cubo en 2 subconjuntos disjuntos de piezas: las piezas que mueve (llamadas *soporte* en teoría de grupos) y las que mantiene fijas (denominados *puntos fijos*). Un giro preservará un bloque si todas las piezas se mantienen en alguno de estos subconjuntos, es decir, si todas las piezas se mantienen inamovibles o si todas se desplazan al unísono en dicho giro. Expresado de manera más matemática, el conjunto de piezas del bloque ha de ser disjunto a uno de los dos subconjuntos, al soporte o elementos fijos de la permutación. Si tuviera elementos en común en ambos subconjuntos, implicaría que el bloque se rompería con el giro. Tras verificar esto, el giro sería válido si esta propiedad se verifica para todos los bloques del cubo. Más formalmente, diremos:

Sean:

- El conjunto de piezas: $P = \{0, 1, \dots, 53\}$
- b_i un bloque: $b_i \subset \wp(P)$
- B el conjunto de bloques del cubo: $B = \{b_1, b_2, \dots, b_n\}$
- σ un giro: $\sigma \in \{R, R', R2, U, \dots, \}$
- Los puntos fijos del giro, es decir, los que no se intercambian:
 $pf(\sigma) = \{x \in P | \sigma(x) = x\}$
- El soporte del giro, es decir, los que si que la cambian:
 $sop(\sigma) = \{x \in P | \sigma(x) \neq x\}$

Diremos que:

$$\sigma \text{ preserva } b \iff (b \cap sop(\sigma) = \emptyset) \vee (b \cap pf(\sigma) = \emptyset) \quad (3.1)$$

$$\sigma \text{ es válido} \iff \sigma \text{ preserva } b_i, \forall b_i \in B \quad (3.2)$$

Ejemplos: Tenemos un cubo en su estado resuelto, definido por el estado:
 $Posicion = [0, 1, \dots, 53]$

$$Bloqueos = \{b_1, b_2\} = \{\{24, 25, 28, 29, 48\}, \{32, 33, 38, 39, 49, 51, 52\}\}$$

¿R es un giro válido? Si se intenta realizar un giro de la capa R, se calculará:

¿R preserva b_2 ?

$$sop(R) = \{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 28, 29, 34, 35, 36, 37, 42, 43, 50\}$$

$$pf(R) = P \setminus sop(R)$$

$$sop(R) \cap b_2 = \emptyset$$

$$pf(R) \cap b_2 = \{32, 33, 38, 39, 49, 51, 52\}$$

b_2 es disjunto con $soporte(R)$, por lo que verifica 3.1. Luego R preserva b_2

¿R preserva b_1 ?

$$sop(R) = \{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 28, 29, 34, 35, 36, 37, 42, 43, 50\}$$

$$pf(R) = P \setminus sop(R)$$

$$sop(R) \cap b_1 = \{28, 29\}$$

$$pf(R) \cap b_1 = \{24, 25, 48\}$$

R no preserva b_1 porque no verifica 3.1. En conclusión, como no preserva todos los bloques, R no es un giro válido en esta posición.

¿ U es un giro válido? Si se intenta realizar un giro de la capa U , se calculará:

¿ U preserva b_1 ?

$$sop(U) = \{9, 10, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 30, 31, 24, 25, 26, 27, 28, 29, 48\}$$

$$pf(U) = P \setminus sop(U)$$

$$sop(U) \cap b_1 = \{24, 25, 28, 29, 48\}$$

$$pf(U) \cap b_1 = \emptyset$$

b_1 es disjunto con los puntos fijos de U . Por ello, verifica 3.1.

¿ U preserva b_2 ?

$$sop(U) = \{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 28, 29, 34, 35, 36, 37, 42, 43, 50\}$$

$$pf(U) = P \setminus sop(U)$$

$$sop(U) \cap b_2 = \emptyset$$

$$pf(U) \cap b_2 = \{32, 33, 38, 39, 49, 51, 52\}$$

b_2 es disjunto con $sop(U)$, por lo que verifica 3.1. Como U preserva todos los bloques del cubo, verifica 3.2, y por lo tanto U es un giro válido en esta configuración.

3.7. Motor de búsqueda

Para realizar las búsquedas, se va a utilizar el algoritmo de Korf [11]. Este es una versión del algoritmo *Iterative-Deepening A** (IDA*) [12] con unas heurísticas muy concretas, basadas en las piezas de un cubo de Rubik. Este es un algoritmo idóneo para nuestro problema, debido a la alta dimensión de nuestro árbol de búsqueda (del orden de $4,3 \cdot 10^{19}$ en un cubo sin bloqueos). Los algoritmos de búsqueda en anchura no nos servirían, dado que se basan en almacenar nodos en alguna estructura de datos, y es altamente probable que nos quedáramos sin memoria en una ejecución. Los algoritmos de búsqueda en profundidad son más eficientes en memoria, pero también cuentan con limitaciones. Podrían entrar en bucles infinitos o devolver soluciones subóptimas, por lo que tampoco nos serían de utilidad. Para ello, los algoritmos de profundidad iterativa son de gran utilidad, puesto que combinan características de ambos planteamientos. Los algoritmos de profundidad iterativa consisten en realizar una búsqueda en profundidad acotando su profundidad máxima, e ir aumentando progresivamente esta cota hasta hallar una solución. La complejidad temporal de este

Capítulo 3. Análisis y diseño

algoritmo para un factor de ramificación b y una solución a profundidad d es de $\mathcal{O}(b^d)$, mientras que su complejidad en memoria es de $\mathcal{O}(d)$. Aunque no podemos reducir esa naturaleza exponencial, sí que podremos reducir su base y por ello el árbol de búsqueda. Estos algoritmos tienen un funcionamiento muy bueno, puesto que terminan encontrando una solución razonable sin abortar por falta de memoria. Dado que cada iteración se realiza mediante búsqueda en profundidad, el algoritmo es muy eficiente en memoria, dado que solo se necesita almacenar la pila de llamadas de la rama que se está explorando en cada momento. La memoria restante puede ser utilizada para almacenar heurísticas. Su desventaja principal consiste en que es muy probable que se recomputen algunos nodos múltiples veces. Por ello, trataremos de que estos cómputos repetidos sean eficientes. No obstante, es un algoritmo con un funcionamiento muy bueno para problemas como este.

3.7.1. Reducción del árbol de búsqueda

Antes de razonar sobre las búsquedas, es interesante generar casos de forma estratégica para buscar sobre un árbol de menor tamaño. Dado que tenemos $6 \cdot 3 = 18$ opciones para realizar un giro, para n giros tendremos un árbol del orden de 18^n estados. Aunque no es posible reducir su naturaleza de crecimiento exponencial, sí que podremos reducir la base de esta. Para ello, podremos generar casos que repitan la menor cantidad de estados posibles. Para ello, utilizaremos lo que se conoce como secuencias canónicas, o *canonical sequences* [3]. Estas son secuencias de giros que cuentan con 2 restricciones, que permiten reducir el árbol de manera muy sencilla. En primer lugar, evitan girar la misma capa 2 veces consecutivas, puesto que no accede a situaciones nuevas. En segundo lugar, evitan generar por duplicado los giros de capas paralelas. Por ejemplo, el efecto de los algoritmos *RL* y *LR* es el mismo, por lo que se generará una única vez. Esto es sencillo de implementar si se almacena la capa que se giró en el anterior giro. Podemos definir una relación de orden arbitraria entre las capas, y una función sencilla que calcule a qué eje pertenece ($R - L$, $U - D$ ó $F - B$). Dada una capa de giro previo, generaremos los movimientos de los otros ejes, y solo las que sean estrictamente mayores (o equivalentemente menores) del mismo eje.

Ejemplo Definimos como relación de orden $R < U < F < L < D < B$

Si hemos girado U como última capa, tendremos en la siguiente generación los movimientos de las capas:

$[R, L, F, B, \mathbf{D}]$

Pero si el último giro fue de la capa D como última capa, tendremos en la siguiente generación:

$[R, L, F, B]$

De esta manera, las secuencias UD no se generan por duplicado.

El número de secuencias canónicas para n giros ha sido calculado en multitud de ocasiones [14], [3], [11]. Este viene dado por la siguiente relación de recurrencia:

$$q(n) = \begin{cases} 12 \cdot q(n-1) + 18 \cdot q(n-2) & n > 2 \\ 18 & n = 1 \\ 18 \left(4 \cdot 3 + \frac{3}{2}\right) = 243 & n = 2 \end{cases}$$

La cantidad de secuencias canónicas de cada número de giros se muestra en la tabla 3.2. En esta se aprecia que hemos reducido el factor de ramificación desde 18 a un promedio de 13.34. De esta forma, los estados del árbol serán del orden de 13.34^n para n giros.

Giros ($q(n)$)	Secuencias canónicas	Factor de ramificación $\left(\frac{q(n)}{q(n-1)}\right)$
1	18	-
2	243	13.5
3	3240	13.33
4	43254	13.35
5	577368	13.34
6	$7,7 \cdot 10^6$	13.34
7	$1,02 \cdot 10^8$	13.34
8	$1,37 \cdot 10^9$	13.34
9	$1,83 \cdot 10^{10}$	13.34
10	$2,44 \cdot 10^{11}$	13.34
11	$3,26 \cdot 10^{12}$	13.34
12	$4,35 \cdot 10^{13}$	13.34
13	$5,81 \cdot 10^{14}$	13.34
14	$7,7 \cdot 10^{15}$	13.34
15	$1,03 \cdot 10^{17}$	13.34
16	$1,3 \cdot 10^{18}$	13.34
17	$1,84 \cdot 10^{19}$	13.34
18	$2,46 \cdot 10^{18}$	13.34
19	$3,29 \cdot 10^{21}$	13.34
20	$4,39 \cdot 10^{22}$	13.34

Cuadro 3.2: Número de secuencias canónicas y factor de ramificación

No obstante, la reducción no termina aquí. Esto es válido para cualquier cubo en el que giren 6 capas, pero algunos bloqueos restringen más el árbol. Es posible que bloqueen por completo algunas capas, por lo que no tiene sentido generarlas para descartarlas siempre. La detección de estas capas es muy sencilla: si un bloque incluye varios centros, ninguna de las capas definidas por esos centros podrá moverse. Por ello, antes de comenzar las búsquedas, calcularemos las capas que sí pueden moverse. Esto permite reducir aún más el factor de ramificación del árbol. Las posibilidades de bloqueos de capas completas, los subgrupos que generan y el factor de ramificación de sus secuencias canónicas se muestran en la tabla 3.3. El cálculo detallado de estos valores se muestra en el primer anexo.

Descripción	Ejemplo	Factor de ramificación
2 capas contiguas	$\langle R, U \rangle$	3
3 capas con 2 opuestas	$\langle R, L, U \rangle$	4.85
3 capas contiguas	$\langle R, U, F \rangle$	6
4 capas con 2 opuestas	$\langle R, L, U, F \rangle$	8.19
6 capas	$\langle R, L, U, D, F, B \rangle$	13.34

Cuadro 3.3: Factor de ramificación promedio de algunos subgrupos

3.7.2. Algoritmo Iterative-Deepening A* (IDA*)

El algoritmo IDA* es un algoritmo heurístico basado en profundidad iterativa. En primer lugar, vamos a definir algunas funciones características de los algoritmos de búsqueda heurísticos. Supongamos que nos encontramos en un momento arbitrario de la búsqueda, y estamos explorando el nodo n . Denominaremos $f(n)$ como la distancia mínima desde el nodo n hasta la solución. Esta se calculará como

$$f(n) = g(n) + h(n)$$

donde $g(n)$ es la distancia que hemos acumulado desde el nodo inicial, y $h(n)$ es la distancia mínima que hay desde el nodo n hasta la solución. Esta es aportada por nuestra función heurística. En nuestro caso, estas distancias serán el número de giros en métrica HTM.

El algoritmo IDA* comienza estableciendo como umbral (δ) la distancia estimada del nodo inicial (n_0), es decir, $h(n_0)$ ($f(n_0) = h(n_0)$ porque $g(n_0) = 0$). Se realiza una búsqueda en anchura con una modificación: solo exploraremos los nodos que podrían estar en este subárbol, es decir aquellos que $f(n) \leq \delta$. Por ejemplo, si buscamos con una profundidad máxima de 4 ($\delta = 4$), y a profundidad 1 ($g(n) = 1$) nos encontramos un nodo que está como mínimo a distancia 4 ($h(n) = 4$) sabremos que como mínimos su distancia total será $f(n) = 4 + 1 = 5 > \delta$, y por tanto sabremos que no estará en esta sub-búsqueda. Si tras realizar esta búsqueda no hemos hallado la solución, aumentaremos el nuevo umbral al menor $f(n)$ que superó el umbral, y proseguiremos con otra búsqueda similar hasta encontrar la solución.

Requisitos de las heurísticas: admisibilidad y monotonía

Para que el algoritmo IDA* funcione correctamente, necesitamos que las funciones heurísticas siempre sean admisibles y monótonas. Una heurística será monótona cuando nunca sobreestima la distancia restante desde un nodo hasta la solución, es decir,

$$h(n) \leq h^*(n), \forall n$$

siendo $h^*(n)$ la distancia real hasta una solución. Una heurística será monótona si, al proseguir con una búsqueda, las distancias estimadas nunca decrecen. Es decir, denominando $s(n)$ a un nodo sucesor de n (es explorado posteriormente), se verifica que

$$f(n) \leq f(s(n)), \forall n$$

Está demostrado que si existe una solución, el algoritmo IDA* devolverá una solución óptima si la heurística que utilicemos es admisible y monótona [12].

3.7.3. Algoritmo de Korf

El algoritmo de Korf [11] es un algoritmo diseñado para resolver el cubo de Rubik (estándar) de forma óptima. Consiste en realizar una búsqueda mediante IDA*, utilizando unas heurísticas muy concretas. Las heurísticas se basan en bases de datos de patrones, donde se almacenan el menor número de movimientos necesarios para resolver algunas piezas. Si quisieramos almacenar el número de movimientos de cada estado, y pudiéramos almacenar en 1 byte cada posición, necesitaríamos 43 millones de terabytes, lo cual es inviable. Es por ello que Korf propuso utilizar varias bases de datos de patrones con algunas piezas. Más concretamente, se utilizan 3 bases de datos: una para las esquinas del cubo, una para 6 aristas y otra para las 6 aristas restantes. De esta manera, cada base de datos tendrá un tamaño razonable. En total, se requerirán 173,336 MB de memoria para almacenarlas, lo cual es más que razonable. Para devolver un valor, consultaremos los movimientos necesarios para resolver cada tipo de pieza, y devolveremos el máximo de estos 3 valores. De esta manera, nos aseguramos que esta heurística sea admisible y lo más informada posible. Si denominamos $m_x(n)$ como los movimientos necesarios para resolver las piezas x del estado n (que se calculan consultando la base de datos de patrones), su función heurística será:

$$h(n) = \max\{m_c(n), m_{e1}(n), m_{e2}(n)\}$$

Para generarlas, será necesario partir desde el estado resuelto y realizar una generación por búsqueda en anchura. Habrá que ir almacenando la profundidad acumulada desde el estado resuelto, hasta que se alcance la profundidad deseada o que no se generen casos nuevos. iremos almacenando para cada posición el número de movimientos acumulados, y será imprescindible que lo guardemos en la primera vez que alcancemos cada estado. Si no hiciéramos esto, estaríamos generando unas heurísticas no admisibles. Esta heurística es admisible y monótona si las generamos correctamente.

3.7.4. Heurísticas de Korf

Tamaño de las bases de datos

Es de utilidad conocer el tamaño de cada base de datos. Como veremos más adelante, seremos capaces de almacenarlas en un vector al que indexaremos según el estado del cubo. Para las esquinas, todas las permutaciones podrán aparecer (aunque no con todas las permutaciones de aristas), pero no ocurre lo mismo con la orientación. La orientación de la última esquina queda determinada por la de las otras 7, por lo que no será necesario guardarla. El número de permutaciones posibles de esquinas será:

$$|v_c| = 8! \cdot 3^7 = 88,179,840$$

En cada subconjunto de 6 aristas, todas las posibilidades pueden aparecer (aunque no de forma simultánea). La orientación una arista queda determinada

Capítulo 3. Análisis y diseño

por la de las otras 11, pero siempre podremos voltear una arista de cada uno de los subconjuntos. El número de posibilidades de ambas bases de datos será el mismo, que será:

$$|v_{e1}| = |v_{e2}| = \frac{12!}{6!} \cdot 2^6 = 42,577,920$$

En total, tendremos que almacenar 173.335.680 números. El máximo de movimientos para resolver un cubo de Rubik es 20 [3], por lo que para guardar los movimientos restantes nos valdrán con vectores de enteros de un byte (con rango de 0-255). Para poder acceder a estos valores, es necesario poder indexarlos de la manera más eficiente posible. Con indexación nos referimos a, dado un cubo, saber a qué posición del vector acceder. Como veremos más adelante, contamos con una forma de numerar las permutaciones, y de esta manera convertir cada posición del cubo en 3 números (uno por cada tipo de pieza). Gracias a esto, podremos almacenar en cada vector el número de movimientos requeridos en la posición determinada por este número. De esta manera, conseguimos guardar todas estas bases de datos en 173,336 MB de memoria, lo cual es un espacio razonable. Este cálculo se muestra en la tabla 3.4.

Piezas	Posibles estados	Bytes por estado	Bytes totales
Esquinas	88.179.840	1	88.179.840
Primeras aristas	42.577.920	1	42.577.920
Últimas aristas	42.577.920	1	42.577.920
Total	173.335.680	1	173,336 MB

Cuadro 3.4: Tamaños de las bases de datos de patrones

En resumen, para calcular la heurística de Korf de un estado arbitrario, primero calcularemos la posición a la que acceder en cada uno de los 3 vectores. A esta función la llamaremos k_x para la pieza x . Al final, tomaremos el valor máximo, es decir:

$$h(n) = \max\{h_c(n), h_{e1}(n), h_{e2}(n)\} = \max\{v_c[k_c(n)], v_{e1}[k_{e1}(n)], v_{e2}[k_{e2}(n)]\}$$

Indexación de las heurísticas de Korf

A continuación se va a explicar cómo se consigue almacenar las heurísticas en un vector por cada base de datos y cómo acceder a estos valores. Para ello, necesitamos una función que, dada una permutación y orientación de piezas concreta, calcule un número, que utilizaremos como posición a utilizar en el vector. Esta función ha de ser biyectiva, para que cada posición de piezas tenga una única posición en el vector. Como mínimo, nos interesa la inyectividad, es decir, no hay 2 elementos que reciban el mismo número. Para calcularlo, podemos utilizar la numeración factorial [23] o la numeración npr, sistemas basados en los códigos de Lehmer. Cada conjunto de permutaciones o variaciones describe un sistema de numeración, sobre el cual podemos calcular el número definido por cada elemento. Tras calcular dicho número, calcularemos otro número en base a la orientación, y por último los combinaremos de forma única.

Sistema de numeración factorial Existe una manera de numerar permutaciones, basada en el sistema de numeración factorial. De esta manera, podemos convertir una permutación de n elementos en un número del rango $[0, n! - 1]$, el cual nos devuelve su posición en orden lexicográfico. Para ello, recorremos los números de la permutación en orden, anotando su posición en la lista ordenada (comenzando en 0 y sin contar los que ya han aparecido). A continuación, se multiplican por su correspondiente posición y se suman (ver tabla 3.5 y ejemplo).

Cifras en numeración factorial	$a_0 :$	$a_1 :$	\dots	$a_{n-1} :$	a_n
Índice	0	1	\dots	$n-1$	n
Número máximo	n	$n-1$	\dots	1	0
Multiplicador	$n!$	$(n-1)!$	\dots	$1!$	$0!$

Cuadro 3.5: Numeración factorial

Ejemplo Vamos a realizar ejemplos para numerar las permutaciones de esquinas entre $[0, 8! - 1] = [0, 40,319]$.

Permutación $[4, 2, 3, 5, 1, 7, 6, 0] \rightarrow 4 : 2 : 2 : 2 : 1 : 2 : 1 : 0$ (cálculo detallado en tabla 3.6).

Permutación restante	Elementos en orden	Posición	Acumulación
$p = [4,2,3,5,1,7,6,0]$	$[0,1,2,3,4,5,6,7]$	4	4
$p = [-,2,3,5,1,7,6,0]$	$[0,1,2,3,-,5,6,7]$	2	4:2
$p = [-,-,3,5,1,7,6,0]$	$[0,1,-,3,-,5,6,7]$	2	4:2:2
$p = [-,-,-,5,1,7,6,0]$	$[0,1,-,-,-,5,6,7]$	2	4:2:2:2
$p = [-,-,-,-,1,7,6,0]$	$[0,1,-,-,-,-,6,7]$	1	4:2:2:2:1
$p = [-,-,-,-,-,7,6,0]$	$[0,-,-,-,-,-,6,7]$	2	4:2:2:2:1:2
$p = [-,-,-,-,-,-,6,0]$	$[0,-,-,-,-,-,-,6,-]$	1	4:2:2:2:1:2:1
$p = [-,-,-,-,-,-,-,0]$	$[0,-,-,-,-,-,-,-]$	2	4:2:2:2:1:2:1:0
$p = [-,-,-,-,-,-,-,-]$	$[-,-,-,-,-,-,-,-]$	0	4:2:2:2:1:2:1:0

Cuadro 3.6: Ejemplo de numeración factorial

Una vez tenemos la numeración, el resultado será multiplicar cada cifra por su correspondiente multiplicador y sumar, es decir,

$$4 \cdot 7! + 2 \cdot 6! + 2 \cdot 5! + 2 \cdot 4! + 1 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + 0 \cdot 0! = 21,899$$

Podemos observar que este algoritmo tiene una complejidad de $\mathcal{O}(n^2)$. Existe un algoritmo en $\mathcal{O}(n)$ basado en la manipulación de bits [24], pero no será implementado en este caso. El algoritmo descrito no supone una ralentización excesiva del código.

Sistema de numeración npr El sistema de numeración factorial puede extenderse para numerar variaciones. No existe un convenio tan claro sobre su nombre, puesto que algunas fuentes lo denominan *numeración npr* o *numeración de permutaciones parciales*. Si tenemos variaciones de n elementos de los

Capítulo 3. Análisis y diseño

que tomamos r de ellos, le podremos asignar un número del rango $[0, \frac{n!}{r!} - 1]$. En el caso de numerar 6 aristas será $[0, \frac{12!}{6!} - 1] = [0, 665279]$. El procedimiento es similar al de la numeración factorial, pero cambian los multiplicadores (ver tabla 3.7 y ejemplo).

Cifras en numeración factorial	$a_0 :$	$a_1 :$...	a_i	...	$a_{r-1} :$
Índice	0	1	...	i	...	$r - 1$
Número máximo	n	$n - 1$...	$n - i$...	$n - r + 1$
Multiplicador	$\frac{(n-1)!}{(n-r)!}$	$\frac{(n-2)!}{(n-r)!}$...	$\frac{(n-1-i)!}{(n-r)!}$...	1

Cuadro 3.7: Numeración npr

Ejemplo Vamos a realizar un ejemplo para numerar las permutaciones de 6 aristas, es decir, $n = 12, r = 6$, y se darán valores del rango $[0, \frac{12!}{6!} - 1] = [0, 665279]$. Veremos más adelante cómo gestionamos la orientación de las piezas.

Para la variación $[11, 0, 5, 10, 8, 9] \rightarrow 11 : 0 : 4 : 8 : 6 : 6$. Se detalla el cálculo en la tabla 3.8:

Variación	Permutación restante	Posición	Acumulación
$p = [11, 0, 5, 10, 8, 9]$	$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$	11	11:
$p = [-, 0, 5, 10, 8, 9]$	$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -]$	0	11:0
$p = [-, -, 5, 10, 8, 9]$	$[-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -]$	4	11:0:4
$p = [-, -, -, 10, 8, 9]$	$[-, 1, 2, 3, 4, -, 6, 7, 8, 9, 10, -]$	8	11:0:4:8
$p = [-, -, -, -, 8, 9]$	$[-, 1, 2, 3, 4, -, 6, 7, 8, 9, -, -]$	6	11:0:4:8:6
$p = [-, -, -, -, -, 9]$	$[-, 1, 2, 3, 4, -, 6, 7, 8, -, -, -]$	6	11:0:4:8:6:6
$p = [-, -, -, -, -, -]$	$[-, 1, 2, 3, 4, -, 6, 7, 8, -, -, -]$.	11:0:4:8:6:6

Cuadro 3.8: Ejemplo de numeración npr

Una vez tenemos la numeración, el resultado será multiplicar cada cifra por su correspondiente multiplicador y sumar, es decir,

$$\frac{1}{(12-6)!} (11 \cdot 11! + 0 \cdot 10! + 4 \cdot 9! + 8 \cdot 8! + 6 \cdot 7! + 6 \cdot 6!) = 612,352$$

Cabe destacar que, aunque existe la manera de revertir estos algoritmos (es decir, dado un número encontrar su permutación o variación), no se van a utilizar en este proyecto y por eso no se detallan.

Numeración en la base definida por la orientación de las piezas La orientación de las piezas que utilizamos para generar heurísticas vienen determinadas por un vector de \mathbb{Z}_3^8 , y dos de \mathbb{Z}_2^6 . Estos vectores definen un número en base 3 y 2 respectivamente. Para numerar las posibles orientaciones de las piezas, podremos calcular el número decimal en esta base de numeración. Podemos hacer una optimización: ya que la orientación de la última esquina viene determinada por la de las otras 7, podremos no almacenarla, y de esta manera utilizaremos un rango 3 veces menor. De esta forma, la orientación de esquinas nos dará un

número del rango $[0, 3^7 - 1] = [0, 2186]$, y cada mitad de las aristas uno del rango $[0, 2^6 - 1] = [0, 63]$.

Ejemplos si la orientación de las piezas de un cubo es (esquinas y dos mitades de aristas respectivamente):

$$(2 \ 1 \ 0 \ 0 \ 2 \ 1 \ 1 \ 2),$$

$$(1 \ 1 \ 0 \ 0 \ 1 \ 1)$$

$$(1 \ 0 \ 0 \ 1 \ 0 \ 0)$$

El número definido por la orientación de las esquinas será (nótese que la última cifra se ha eliminado):

$$2100211_3 = 2 \cdot 3^6 + 1 \cdot 3^5 + 0 \cdot 3^4 + 0 \cdot 3^3 + 2 \cdot 3^2 + 1 \cdot 3^1 + 1 \cdot 3^0 = 1,723$$

Y el de cada mitad de aristas será:

$$110011_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 51$$

$$100100_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 36$$

Numeración de estados. Unificación de permutación y orientación Una vez hemos numerado la permutación y orientación por separado de cada tipo de pieza de cada estado, necesitamos combinar ambos valores en un único número para cada tipo de pieza. Para ello, podremos calcular:

$$k = p(1 + m_o) + o$$

Siendo k la nueva clave, p, o a las numeración de permutación y orientación de un tipo de pieza, y m_o al valor máximo posible para la orientación. Esto es análogo a, si tenemos 2 cifras decimales d_1d_2 , del rango $[0,9]$, podemos combinarlas mediante $10d_1 + d_2$, lo cual nos dará una numeración única. De esta forma, podremos calcular las claves de esquinas y de 6 aristas respectivamente mediante:

$$k_c = 3^7 \cdot p_c + o_c$$

$$k_e = 2^6 \cdot p_e + o_e$$

El valor máximo de esquinas será:

$$3^7 \cdot (8! - 1) + 3^7 - 1 = 3^7 \cdot (8! + 1 - 1) - 1 = 8! \cdot 3^7 - 1$$

Y el de aristas será:

$$2^6 \cdot \left(\frac{12!}{6!} - 1 \right) + 2^6 - 1 = 2^6 \cdot \left(\frac{12!}{6!} - 1 + 1 \right) - 1 = 2^6 \cdot \frac{12!}{6!} - 1 =$$

Si nos fijamos, el valor mínimo para estos valores es siempre 0, y el máximo es precisamente el total de opciones posibles de cada pieza restándole 1. Como

Capítulo 3. Análisis y diseño

Pieza	Permutación	Orientación	Combinación de ambos
Esquinas	$8! - 1 = 40,319$	$3^7 - 1 = 2,186$	$8! \cdot 3^7 - 1 = 88,179,839$
Aristas (6)	$\frac{12!}{6!} - 1 = 665,280$	$2^6 - 1 = 63$	$2^6 \cdot \frac{12!}{6!} - 1 = 42,577,919$

Cuadro 3.9: Resumen de valores máximos de indexación por piezas

esta función es biyectiva, hemos conseguido una numeración perfecta para cada tipo de pieza. En la tabla 3.9 se recogen los valores máximos de indexación para cada tipo de pieza. (el mínimo es 0 en todos los casos).

Ejemplo Sea el estado de un cubo arbitrario, definido por sus vectores de esquinas

$$(7 \ 1 \ 2 \ 0 \ 3 \ 5 \ 6 \ 4)$$

$$(2 \ 0 \ 0 \ 1 \ 2 \ 0 \ 0 \ 1)$$

$\in S_8 \times \mathbb{Z}_3^8$ y sus vectores de aristas:

$$(0 \ 1 \ 2 \ 4 \ 8 \ 3)$$

$$(0 \ 0 \ 0 \ 1 \ 1 \ 1)$$

$$(6 \ 7 \ 5 \ 9 \ 10 \ 11)$$

$$(0 \ 0 \ 1 \ 0 \ 0 \ 0)$$

$$\in P(12,6) \times \mathbb{Z}_2^6 \times P(12,6) \times \mathbb{Z}_2^6$$

Vamos a calcular las posiciones a las que acceder para modificar o consultar sus heurísticas.

Esquinas

Para calcular el índice de la permutación, utilizamos la numeración factorial.

$$(7 \ 1 \ 2 \ 0 \ 3 \ 5 \ 6 \ 4) \rightarrow 7 : 1 : 1 : 0 : 0 : 1 : 1 : 0 \rightarrow 7 \cdot 7! + 6! + 5! + 2! + 1! = 36,123$$

Después, calculamos la posición de la orientación de las esquinas:

$$(2 \ 0 \ 0 \ 1 \ 2 \ 0 \ 0 \ 1) \rightarrow 2 \cdot 3^2 + 3^3 + 2 \cdot 3^6 = 1,503$$

Combinando los valores, obtenemos la posición en el vector de esquinas, es decir, $36,123 \cdot 3^7 + 1,503 = \mathbf{79.002.504}$

Ahora, podemos calcular la clave de la primera mitad de las aristas. Permutación, mediante numeración npr:

$$(0 \ 1 \ 2 \ 4 \ 8 \ 3) \rightarrow 0 : 0 : 0 : 1 : 4 : 0 \rightarrow 84$$

Orientación:

$$(0 \ 0 \ 0 \ 1 \ 1 \ 1) \rightarrow 1 + 2 + 4 = 7$$

Combinando los resultados, queda $2^6 \cdot 84 + 7 = \mathbf{5.381}$

Procedemos de forma similar con la segunda mitad de aristas.

$$(6 \ 7 \ 5 \ 9 \ 10 \ 11) \rightarrow 6 : 6 : 5 : 6 : 6 : 6 \rightarrow 365,784$$

$$(0 \ 0 \ 1 \ 0 \ 0 \ 0) \rightarrow 365,784 \cdot 2^6 + 8 = \mathbf{23.410.184}$$

Capítulo 4

Desarrollo e implementación

4.1. Entrada de los datos

El input para definir un cubo tendrá varias partes. Por comodidad, el usuario puede comenzar estableciendo unos alias para cada cara. Por ejemplo, podrá introducir las pegatinas por sus iniciales (en inglés son todas diferentes) en lugar del color completo en cada una de las 54. Posteriormente, se pedirá definir el patrón de colores, introduciendo los colores de cada cara. Se realizará en hexadecimal o en algunas abreviaturas aportadas por *Manim*, aunque habrá un patrón de colores por defecto para que el usuario no tenga que introducirlo. Posteriormente, introducirá los colores de las pegatinas de cada cara, según el orden definido por 3.4. El programa revisará que hay exactamente 9 pegatinas válidas, y que la pegatina central cuadra con la de la capa que está introduciendo. Si no se cumple, pedirá la capa otra vez. Después, el usuario introduce los bloqueos. Define cada pieza por las capas a las que pertenece separadas por un guion, y juntará varias piezas con el símbolo +. En cualquier momento, si el usuario deja vacío el campo, se guardará el valor por defecto en caso de haberlo, a excepción de la introducción de los bloques, que indica haber finalizado de introducirlos. Un ejemplo de input sin errores se muestra a continuación:

```
#SELECCIÓN DE ALIAS

Insert alias for U face (default: w)
b
Insert alias for R face (default: r)

Insert alias for F face (default: g)
k
Insert alias for D face (default: y)
g
Insert alias for L face (default: g)
```

Capítulo 4. Desarrollo e implementación

Insert alias for B face (default: b)
y

#DEFINICIÓN DEL PATRÓN DE COLORES

Default hexadecimal colours:
White: #FFFFFF
Red: #B90000
Green: #009B48
Yellow: #FFD500
Orange: #FF5900
Blue: #0045AD

Insert colour for U face (alias b)
(hexadecimal or by name, default #FFFFFF (white))
#92A8D1
Insert colour for R face (alias r)
(hexadecimal or by name, default #B90000 (red))

Insert colour for F face (alias k)
(hexadecimal or by name, default #009B48 (green))
#282828
Insert colour for D face (alias g)
(hexadecimal or by name, default #FFD500 (yellow))
#CCE5CC
Insert colour for L face (alias o)
(hexadecimal or by name, default #FF5900 (orange))

Insert colour for B face (alias y)
(hexadecimal or by name, default #0045AD (blue))
#FDB510

#INTRODUCCIÓN DE LAS PEGATINAS CARA A CARA

Insert U (alias b) face colours, separated by spaces
(default: b b b b b b b b b b)
b b b b b b k o o
Insert R (alias r) face colours, separated by spaces
(default: r r r r r r r r r r)
b r r b r r k g g
Insert F (alias k) face colours, separated by spaces
(default: k k k k k k k k k k)
r k k k k k k k r

```

Insert D (alias g) face colours, separated by spaces
                                              (default: g g g g g g g g g )
g g b r g o r g o
Insert L (alias o) face colours, separated by spaces
                                              (default: o o o o o o o o o )
o o g o o r g g o
Insert B (alias y) face colours, separated by spaces
                                              (default: y y y y y y y y y )

```

#DEFINICIÓN DE LOS BLOQUES

```

(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
y+b+o
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
k+k-o-b
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
r+r-y
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
g+g-y
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
r-y-b+r-b
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
y-g-o+g-o
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)
y-g-r+g-r
(Optional) insert a block in the format: c1-c2-c3+c1-c2+c1
                                              (empty for finish)

```

Este input habrá que convertirlo a la codificación de *Haskell*, para lo que se realiza un reordenamiento y una conversión a números. Tendrá que decidir estos números según las posiciones relativas de 2 pegatinas de arista o 3 de esquina, para lo que se realiza una pequeña búsqueda.

4.2. Salida

En caso de haber solución al input introducido, el input devuelve la secuencia de movimientos que resuelve el cubo mediante la notación habitual (ver 2.1.2). Esto viene acompañado de la visualización.

Solution found: D' F' D R' D2 R D F D2 F' D'

4.3. Visualización

Se ha realizado una interfaz para visualizar un cubo en perspectiva isométrica y poder comprobar el input. Esta genera un vídeo de un cubo en formato mp4. En el vídeo, se muestra un cubo que gira sobre una diagonal, lo que permite visualizarlo alrededor de sus 6 caras. Posteriormente, aplica al input los giros hasta terminar con el estado resuelto. Algunos fotogramas del resultado se muestran en las figuras 4.1 y 4.2 respectivamente.

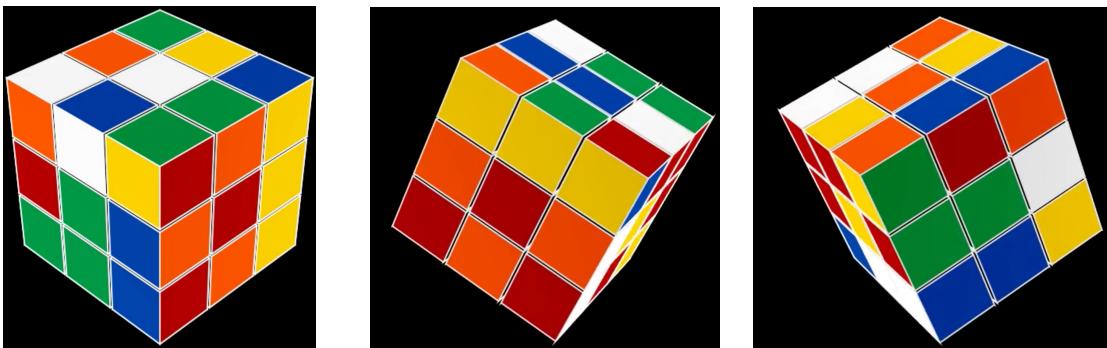


Figura 4.1: Visualización de un cubo rotando sobre una diagonal

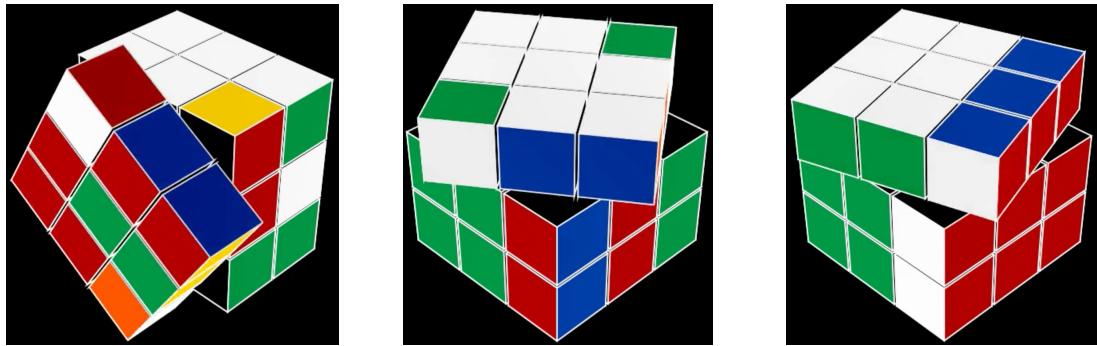


Figura 4.2: Visualización de un cubo siendo resuelto

Estos vídeos se han realizado con la herramienta *Manim*, que funciona sobre *Python*. Para ello, se ha realizado un código de plantilla, al que se le pasan los argumentos como parámetro. Esto facilita su llamada desde la terminal u otros programas, incluyendo los de *Haskell*. Los argumentos que se pasan son el tiempo en segundos de rotación sobre la diagonal, el tiempo de cada movimiento de una capa, la descripción de a qué capa pertenece cada pegatina (en el orden de 3.8), la lista de colores de cada cara definida por sus colores en hexadecimal u otras abreviaturas de *Manim* (en orden U, R, F, D, L, B) y el algoritmo a ejecutar. Un resumen del formato de estos argumentos se muestra en la tabla 4.1.

Argumento	Formato
Tiempo de rotación	Numérico (segundos)
Tiempo de cada giro	Numérico (segundos)
Calidad	String: "low_quality" o "high_quality"
Estado de un cubo	Pegatinas en orden
Patrón de colores	String con 6 colores
Algoritmo	String (secuencia de movimientos)

Cuadro 4.1: Argumentos del programa del visualizador

Un ejemplo de llamada a este programa desde la terminal de comandos sería:

```
~$ python manim_cube_visualizer.py
10 1.5 "low_quality"
"UUUUUUUUURRRRRRRRRFFFFFFFFDDDDDDDDLLL LLLLBBBBBBBBB"
"WHITE, #B90000, #009B48, #FFD500, #FF5900, #0045AD"
"R' U F2"
```

El código de plantilla se muestra a continuación. El método *init* instancia un objeto de tipo cubo con sus valores necesarios. Esta cuenta con valores de argumentos por defecto (como el patrón de colores o el estado resuelto), aunque a lo mejor no es posible generar este código desde terminal sin asignar estos valores. El método *construct* ubica el cubo en la pantalla y realiza la animación. En primer lugar, coloca el cubo en el centro de la pantalla y ajusta la posición de la cámara para observarlo en perspectiva. Posteriormente, rota sobre la diagonal. A continuación, ejecuta los movimientos pasados como argumento. Por último, rota sobre la diagonal una vez más, para que el usuario pueda comprobar el estado final.

```
manim_cube_visualizer.py

class Plantilla(ThreeDScene):

    def __init__(self, algorithm,
                 initial_state="UUUUUUUUURRRRRRRRRFFFFFFFF
                               DDDDDDDDDLLL LLLLBBBBBBBB",
                 color_scheme = [WHITE, "#B90000", "#009B48",
                                 "#FFD500", "#FF5900", "#0045AD"],
                 time_of_rotation = 2, time_of_turns = 1, **kwargs):
        super().__init__(**kwargs)

    """PARTE 1: CREACIÓN DEL CUBO
    Y SU CONFIGURACIÓN ORIGINAL"""

        self.cube = RubiksCube(colors=color_scheme).scale(0.7)
```

Capítulo 4. Desarrollo e implementación

```
    self(cube.set_state(initial_state)

    self.algorithm = algorithm
    self.time_of_rotation = time_of_rotation
    self.time_of_turns = time_of_turns

def construct(self):
    """PARTE 2: COLOCACIÓN DEL CUBO EN PANTALLA"""

    # Configurar posiciones y
    # cámara en perspectiva isométrica
    self(cube.move_to(ORIGIN)
    self.move_camera(phi=54.75*DEGREES,
                      theta=(45+180)*DEGREES)
    self.renderer.camera.frame_center =
        self(cube.get_center())

    # Mostrar por pantalla con animación "FadeIn"
    self.play(FadeIn(self(cube)))
    c = self(cube.cubies[0,0,0]

    self.wait(2)

    #Rotar alrededor de la diagonal
    self.play(Rotate(self(cube),
                     angle=2*PI, axis=[-1,-1,-1],
                     run_time=self.time_of_rotation, rate_func=linear))

    self.wait(1)

    """PARTE 3: EJECUTAR LOS MOVIMIENTOS
    QUE RESUELVEN EL CUBO"""

    list_of_moves = self.algorithm.split(" ")
    #Delete empty moves
    list_of_moves = [s for s in list_of_moves if s.strip()]

    for m in list_of_moves:
        t = self.time_of_turns
        if m[-1] == '2':
            t = 1.75 * self.time_of_turns
            self.play(CubeMove(self(cube, m), run_time=t)

    #Terminar rotando sobre la diagonal
    self.play(Rotate(self(cube,
                      angle=2 * PI, axis=[-1, -1, -1],
```

```

    run_time=self.time_of_rotation/2, rate_func=linear) )

#Desvanecer
self.wait(2)

self.play(FadeOut(self(cube) )

""" GENERACIÓN DEL VÍDEO: """
#Procesado de argumentos:
config.quality = "low_quality"

arg_tRotation = int(sys.argv[1])
arg_tMove = float(sys.argv[2])
arg_quality = sys.argv[3]
config.quality = arg_quality
arg_initial_state = sys.argv[4]
list_colours = sys.argv[5].split(",")
arg_moves = sys.argv[6]

#Creación y renderizado del video
sc = Plantilla(initial_state=arg_initial_state,
                algorithm=arg_moves,
                color_scheme=list_colours,
                time_of_rotation=arg_tRotation,
                time_of_turns= arg_tMove)

sc.render(preview=True)

```

4.4. Implementación de los datos

4.4.1. Cubo de Rubik estándar

Como se discutió en 2.4, el cubo de Rubik tiene una estructura algebraica de grupo. Esto nos permite implementar un cubo en *Haskell* de manera muy concisa, utilizando el módulo *Data.Group* [25]. A nivel de representación, como se especificó en 3.6.1, numeraremos cada pegatina con un número entre 0 y 53, y cada estado se representará mediante un vector con estos números en un orden correspondiente. Cabe destacar que estaremos definiendo la operación entre 2 estados del cubo como colocar las piezas del 1º elemento en el orden indicado por el 2º. Se utilizan vectores *Unboxed* por motivos de eficiencia, puesto que están basados en regiones adyacentes de memoria y son idóneos para tipos básicos. Esta mejora logra alrededor de un 10% de mejora en tiempo.

```

--importamos módulos de datos
import Data.Group
import qualified Data.Vector.Unboxed as V

```

Capítulo 4. Desarrollo e implementación

```
--Definimos el tipo Cube como un vector de enteros
newtype Cube = Cube (V.Vector Int) deriving

--definimos la operación binaria
instance Semigroup Cube where
  (Cube v1) <>> (Cube v2) = Cube (V.backpermute v1 v2)

--definimos el elemento neutro
instance Monoid Cube where
  mempty = Cube (V.fromList [0..53])

--definimos la forma de invertir una permutación
instance Group Cube where
  invert (Cube xs) =
    Cube (V.fromList (invert_perm (V.toList xs)))

--Funciones auxiliares
invert_perm :: [Int] -> [Int]
invert_perm xs = map fst tups_ord
  where
    neutral = [0 .. (length xs - 1)]
    tups = zip (neutral) xs
    tups_ord = sort_by_snd tups

sort_by_snd :: Ord b => [(a, b)] -> [(a, b)]
sort_by_snd xs = sortBy (compare `on` snd) xs
```

A diferencia de los cubos con bloqueos, un cubo estándar siempre puede girar. Para codificar uno bandaged, necesitamos comprobar si cada giro puede ser realizado. Esto último depende de la codificación de los giros, por lo que se detalla con anterioridad.

4.4.2. Giros

Podemos codificar un giro como la capa que se gira y el número de giros realizados. Lo vamos a codificar en base a cuartos de giro, es decir, el número de giro podrá ser $\{0, 1, 2, 3\}$.

```
data Face = N | R | U | F | L | D | B

--Representamos un giro mediante su capa y
--los cuartos de vuelta que se realizan.
newtype Turn = Turn(Face, Int) deriving (Eq)
```

Necesitamos conocer qué piezas intercambia cada movimiento. Ahora se muestran solo un giro, pero en el código se replica para las 18 opciones.

```

permOfTurn :: Turn -> Cube
permOfTurn (Turn(R, 1)) = Cube [0,1,2,3,4,5,11,9,10,13,14,
                                12,17,15,16,7,8,6,18,19,20,21,22,23,24,25,
                                26,27,34,35,30,31,32,33,42,43,28,29,38,39,
                                40,41,36,37,44,45,46,47,48,49,50,51,52,53]

(...)
```

Tras esto, podemos definir el tipo *Algorithm*, el cual es una lista de movimientos. El conjunto de algoritmos con la operación concatenación (y simplificación) forma un grupo.

La inversión de un movimiento es girar la misma capa en sentido inverso (es decir, para invertir n cuartos de vuelta, realizamos un giro de $(4 - n)$). Para invertir un algoritmo, vamos invirtiendo los movimientos en sentido inverso, es decir, podemos realizar $[alg, giro]^{-1} = [giro^{-1}] \cdot [alg]^{-1}$, donde \cdot representa la concatenación de movimientos. Esto se calcula de manera recursiva hasta terminar los movimientos, puesto que el inverso del movimiento nulo es el movimiento nulo. Para calcular la permutación de un algoritmo, realizamos los giros en orden partiendo de su estado resuelto.

También forma un grupo, el cual podemos modelizar como:

```

--Definimos el tipo como lista de giros
data Algorithm = Algorithm [Turn]

--Definimos la operación binaria (concatenar y simplificar)
instance Semigroup Algorithm where
    (Algorithm xs1) <> (Algorithm xs2) =
        Algorithm (simplifyTurns (xs1 ++ xs2))
    --No se muestran detalles de la simplificación

--Definimos el elemento neutro (ningún giro)
instance Monoid Algorithm where
    mempty = Algorithm []

--Definimos la inversión de un algoritmo
--Análogo a inversión de palabra
instance Group Algorithm where
    invert (Algorithm (xs)) =
        Algorithm (invertListofTurns xs)

--Funciones auxiliares
invertListofTurns :: [Turn] -> [Turn]
invertListofTurns [] = []
invertListofTurns (x : xs) =
    invertListofTurns xs ++ [invertTurn x]

--Invertir un único giro
```

Capítulo 4. Desarrollo e implementación

```
invertTurn :: Turn -> Turn
invertTurn (Turn(f, n)) = Turn (f, (4-n))

--Calcula la permutación que realiza un algoritmo
algToPerm :: Algorithm -> Cube
algToPerm (Algorithm xs) = mconcat (map permOfTurn xs)
```

4.4.3. Cubo con restricciones

Como discutimos en 3.6.1, un cubo con bloqueos lo podemos modelizar como un cubo estándar al que le añadimos las piezas bloqueadas. Cada bloque será un conjunto numérico, puesto que codificamos cada pegatina con un número. Una versión simplificada de código en *Haskell* sería la siguiente:

```
import qualified Data.Set as S

--Codificamos como cubo y restricciones
data BandagedCube = BandagedCube
    { stdCube :: Cube,
    restrictions :: S.Set (S.Set Int) }
```

La función que lo comprueba se denomina *tryToTurn*, que devuelve un elemento de tipo *Maybe BandagedCube*. En caso de ser un movimiento válido, solo modifica la lista de piezas y se empaqueta con *Just*. En caso contrario, devuelve *Nothing*. Una simplificación del código es la siguiente:

```
--Hace un giro cuando no rompe ningún bloque
tryToTurn :: BandagedCube -> Turn -> Maybe BandagedCube
tryToTurn bCube currTurn
| validTurn bCube f =
  Just (BandagedCube
    { stdCube = newPerm, restrictions = restr })
| otherwise = Nothing
where
  (BandagedCube currCube restr) = bCube
  (Turn(f, _)) = currTurn
  newPerm = currCube <> (permOfTurn currTurn)

--Calcula si un giro no rompe ningún bloque
validTurn :: BandagedCube -> Face -> Bool
validTurn bCube face = all (checkOneBlock) allRestr
where
  allRestr = restrictions bCube
  checkOneBlock = turnPreserveBlock bCube face

--Calcula si un giro no rompe un bloque concreto
turnPreserveBlock ::
```

4.4. Implementación de los datos

```
BandagedCube -> Face -> S.Set Int -> Bool
turnPreserveBlock (BandagedCube cubeState _) face block =
  (S.disjoint block s1Real) || (S.disjoint block s2Real)
  --Todas las piezas del bloque
  --han de ser permutadas o fijas
where
  (s1, s2) = divideTurn face

  --Calcula las posiciones
  --que mueve (y convierte a conjunto)
  s1RealVec = slicePieces cubeState (S.toList s1)
  s2RealVec = slicePieces cubeState (S.toList s2)
  s1Real = (S.fromList . V.toList) s1RealVec
  s2Real = (S.fromList . V.toList) s2RealVec

  --Divide las piezas en las permutadas y fijas
  divideTurn :: Face -> (S.Set Int, S.Set Int)
  divideTurn m = (piecesAffected m, piecesNotAffected m)

  --Piezas que mueve un giro
  piecesAffected :: Face -> S.Set Int
  piecesAffected R = S.fromList [6,7,8,9,10,11,12,
                                  13,14,15,16,17,28,29,
                                  34,35,36,37,42,43,50]
  piecesAffected U = S.fromList [9,10,11,0,1,2,3,
                                  4,5,6,7,8,30,31,
                                  24,25,26,27,28,29,48]
  piecesAffected F = S.fromList [22,23,21,2,0,1,10,
                                  11,9,14,12,13,33,32,
                                  41,40,31,30,35,34,49]
  piecesAffected L = S.fromList [5,3,4,19,20,18,23,
                                  21,22,1,2,0,38,39,
                                  24,25,46,47,32,33,52]
  piecesAffected D = S.fromList [21,22,23,12,13,14,15,
                                  16,17,18,19,20,46,47,
                                  40,41,42,43,44,45,53]
  piecesAffected B = S.fromList [8,6,7,16,17,15,20,
                                  18,19,4,5,3,37,36,
                                  45,44,27,26,39,38,51]
  piecesAffected _ = S.fromList []
```

Procesamiento y compleción de bloques Este paso solamente se realiza al introducir un nuevo cubo bandaged. El primer paso es preprocesar el conjunto de bloqueos, donde uniremos los bloques con algún elemento en común. Para ello, necesitaremos ir conjunto a conjunto, combinándolos con los que hayamos acumulado que sean disjuntos entre sí. Partiremos desde ningún el conjunto $\{\emptyset\}$ como acumulador. Este cálculo lo realiza la función *genUnions*, que selecciona

Capítulo 4. Desarrollo e implementación

un bloqueo y divide el conjunto acumulado entre los que tienen elementos en común y los que no. Posteriormente, se une el bloqueo que seleccionamos con todos aquellos que fueran no disjuntos, los disjuntos se mantienen intactos y se prosigue hasta finalizar. Una versión simplificada del código se muestra a continuación:

```
--Crear un BandagedCube a partir de un cubo y unas restricciones
newBandagedCube :: Cube -> [[Int]] -> BandagedCube
newBandagedCube cubeOrigin blocks =
    BandagedCube {stdCube = cubeOrigin,
                  restrictions = postProcessedBlocks}
  where
    postProcessedBlocks =
      (expandBlocks . canonicSets)
        (doubleListToDoubleSet blocksSet)

--Calcular uniones de los conjuntos no disjuntos
canonicSets :: S.Set (S.Set Int) -> S.Set (S.Set Int)
canonicSets ss = genUnions (S.toList ss) (listToDoubleSet [[]])

--Auxiliar de canonicSets.
genUnions :: [S.Set Int] -> S.Set (S.Set Int) -> S.Set (S.Set Int)
genUnions [] xs = S.delete (S.empty) xs
genUnions (s0:rest) temp =
  genUnions rest (S.insert unitedSet notCombining)
  where
    (notCombining, toComb) =
      S.partition (S.disjoint s0) temp
    unitedSet = S.unions (S.insert s0 toComb)
```

Tras haber realizado el preprocesamiento, queda calcular las piezas que son indivisibles de cada bloque. Para ello, es necesario calcular para cada bloque y para cada giro válido qué piezas se mantienen unidas. Tras esto, realizaremos las intersecciones pertinentes de los conjuntos. Se muestra una versión simplificada de la función *expandBlocks* y las funciones auxiliares necesarias.

```
--Expansión de todos los bloques
expandBlocks :: S.Set (S.Set Int) -> S.Set (S.Set Int)
expandBlocks setOfBlocks = S.map expand1Block setOfBlocks

--Expansión de un solo bloque
expand1Block :: S.Set Int -> S.Set Int
expand1Block block = pieceIntersections (expansions)
  where
    expansions = map (expandBlockTurn block) [R, U, L, D, B, F]

--Expansión de un bloque y una capa
expandBlockTurn :: S.Set Int -> Face -> S.Set Int
```

```

expandBlockTurn block move
| isBlockPreserved = rightSubSet
| otherwise = s0_53
where
  (xs1, xs2) = divideTurn move
  isBlockPreserved = (S.disjoint block xs1)
    || (S.disjoint block xs2)
  rightSubSet = if (S.disjoint block xs2) then xs1 else
    xs2

pieceIntersections :: (Foldable f) => f (S.Set Int) -> S.Set Int
pieceIntersections = foldl' S.intersection s0_53

--Conjunto de números 0..53
s0_53 :: S.Set Int
s0_53 = S.fromList [0..53]

```

4.5. Motor de búsqueda

4.5.1. Algoritmo de Korf

Se va a hacer un buscador de la manera más genérica posible. Esta función implementa el algoritmo IDA*, que a su vez llama a las funciones necesarias para ejecutar búsquedas generales en profundidad (con las modificaciones de agregar profundidad máxima y heurísticas). De esta forma, se puede utilizar la búsqueda genérica para implementar el algoritmo de Korf utilizando sus heurísticas. El algoritmo general tiene como argumentos el estado inicial, un predicado que defina la propiedad que se está buscando, la lista de giros que se usan para buscar y la función heurística. El programa llamante es responsable de asegurarse que esta heurística sean admisible y monótona. Esta generalización facilitará las estrategias de búsqueda en varios pasos, aunque no se utilizan en este proyecto. Si no se cuentan con heurísticas, se puede pasar una función constante que devuelva 0 siempre (const 0). El argumento de los giros posibles permitirá explorar un árbol de búsqueda de menor factor de ramificación, como se discutió en 3.7.1. Además, esta función de búsqueda genérica solamente genera secuencias canónicas. La cabecera de esta función genérica es:

```

genericSearch :: 
  BandagedCube           -- ^ Initial state
  -> (BandagedCube -> Bool)  -- ^ Condition of the search
  -> [Turn]                -- ^ Turns to generate a new
  state
  -> (BandagedCube -> Int)   -- ^ Heuristic (must be
  admissible)
  -> Maybe Algorithm       -- ^ The solution

```

```
genericSearch ini cond validMoves h
```

Capítulo 4. Desarrollo e implementación

```
--Solucion encontrada:  
| found search = Just (Algorithm (solution search))  
  
--Solucion no encontrada:  
| otherwise = Nothing  
where  
  validLs = nub (map (\(Turn(f,_)) -> f) validMoves)  
  initialSS = SearchingState  
    {found = False, initialState = ini,  
     currentDepth = 0, maximumDepth = h ini,  
     condition = cond,  
     solution = [], validLayers = validLs,  
     listMoves = validMoves,  
     lastFace = N,  
     heuristic = h,  
     minimumExceding = maxBound :: Int}  
--Búsqueda mediante idaStar  
search = idaStar initialSS
```

La función de búsqueda genérica llama a idaStar, que implementa el algoritmo IDA*. Primero calcula los valores iniciales de la búsqueda, y posteriormente, procesa su resultado para devolver un posible algoritmo. Esto simplifica su uso desde el programa llamante. La función idaStar queda de la siguiente manera:

```
idaStar :: SearchingState -> SearchingState  
idaStar initSS  
| found thisSearchSS = thisSearchSS      --Solución hallada  
| (nextDepth > threshold) =             --Siguiente iteración  
  idaStar nextSS  
| otherwise = initSS                  --No existe solución  
where  
  --Búsqueda por dfs  
  thisSearchSS = dfsSgle initSS  
  
  --Profundidad máxima actual  
  threshold = maximumDepth initSS  
  
  --Siguiente profundidad e iteración  
  nextDepth = minimumExceding thisSearchSS  
  nextSS = initSS {maximumDepth = nextDepth}
```

A partir de ahora, todas funciones de búsqueda (IDA* y BFS) utilizarán una estructura para simplificar el código. Esta cuenta con toda la información necesaria para realizar búsquedas. Más concretamente, almacena si encontró una solución, el estado inicial, la profundidad en cada momento, la profundidad máxima establecida, la condición booleana que determina si hemos hallado una solución, los movimientos y capas que se pueden utilizar para buscar, la última capa que se movió (para generar secuencias canónicas), la función heurística y

el menor valor que excedió nuestro máximo umbral. De este modo, se simplifica el código y se aporta más flexibilidad para agregar nuevos datos. La estructura queda de la siguiente manera:

```
data SearchingState = SearchingState {
    found :: Bool,
    initialState :: BandagedCube,
    currentDepth :: Int,
    maximumDepth :: Int,
    condition :: (BandagedCube -> Bool),
    solution :: [Turn],
    validLayers :: [Face],
    listMoves :: [Turn],
    lastFace :: Face,
    heuristic :: (BandagedCube -> Int),
    minimumExceding :: Int}
```

Para implementar las funciones de búsqueda en profundidad, se han realizado 2 funciones. La función *dfsSgle* decide si explorar o no el nodo, calcula los movimientos que podrían ser útiles desde el estado en el que se encuentra y recompone los valores necesarios de los obtenidos en varias ramas posteriormente. La función *dfsMult* recorre recursivamente las ramas y tras encontrar la solución anota el movimiento que realizó para reconstruirla. Estas funciones se llaman alternadamente. Recordemos que no es necesario comprobar en *dfsMult* si el giro es válido (en el sentido de preservar los bloques), puesto que *dfsSgle* descarta estos movimientos. Esto consigue realizar los mismos cálculos con funciones menos costosas, además de realizar copias de movimientos en memoria a la hora de pasar parámetros entre funciones. Descartar los giros no válidos previamente logra reducir alrededor de un 30% en tiempo y memoria total utilizada.

```
--Búsqueda desde un nodo
dfsSgle :: SearchingState -> SearchingState
dfsSgle initialSS
  --solución encontrada
  | predicate ini =
    initialSS {found = True}

  --no explorar, excede de umbral
  | currD > maxD || (currD + h ini > maxD) =
    prunedSS

  --intermedio, seguir buscando
  | otherwise =
    dfsMult initialSS movesToIterate
  where
    (SearchingState _ ini currD maxD predicate _ _
      movesValid 1stFace h exc) = initialSS
    estimLength = currD + h ini
```

Capítulo 4. Desarrollo e implementación

```
--Actualización del mínimo que excede el umbral
prunedSS = if ((estimLength > maxD) && (estimLength <
    exc))
then
    (initialSS{minimumExceeding = estimLength})
else
    initialSS

--Cálculo de los movimientos útiles y válidos
movesToIterate = filter
    (predValidCanonicSequence ini 1stFace)
    movesValid

predValidCanonicSequence :: BandagedCube -> Face -> Turn -> Bool
predValidCanonicSequence bc lsface (Turn(f,_)) =
    ((axisOfFace f /= axisOfFace lsface) || (f > lsface)) &&
    (validTurn bc f)

--Búsqueda de varias ramas
dfsMult :: SearchingState -> [Turn] -> SearchingState

--Ramas finalizadas
dfsMult initialSS [] = initialSS

--Prueba de una rama
dfsMult initialSS (x:xs)

--Rama correcta, recomponer solución
| found thisBrach =
    thisBrach {solution = x : (solution thisBrach) }

--Rama incorrecta, seguir buscando
| otherwise =
    dfsMult (initialSS
        {minimumExceeding =
            min exc0 (minimumExceeding thisBrach) }) xs
where
    (SearchingState _ ini currD maxD
        _____ exc0) = initialSS
    nextState = tryToTurn ini x
    (Turn(lastFaceExecuted, _)) = x

--Cálculo de dfssSgle actualizando algunos valores
thisBrach = dfssSgle (initialSS
    {initialState = fromJust nextState,
    currentDepth = currD + 1,
```

```
lastFace = lastFaceExecuted,
minimumExceding = exc0 })
```

Una vez tenemos un algoritmo de búsqueda genérica, podemos implementar el algoritmo de Korf. Para ello, necesitamos calcular algunos argumentos. La condición que se busca es un predicado que devuelve verdadero si y solo si está en la posición resuelta. La lista de giros estará formada por las tres opciones de giro de cada capa que pueda moverse (90 grados en cada sentido y 180 grados). Para calcular las capas que pueden moverse, eliminaremos de las 6 posibles aquellas cuyo centro esté contenido en algún bloque que englobe al menos 2 centros. La función heurística que pasemos como argumento será la que calcule la heurística de Korf. Una versión simplificada es la siguiente:

```
-- Encuentra una solución óptima con el algoritmo de Korf.
-- Solo utiliza capas no bloqueadas
smartKorfSolver :: BandagedCube -> Maybe Algorithm
smartKorfSolver bc =
    genericSearch bc solvedBC
    (notBlockedMoves bc) korfHeuristic

--Not blockedMoves utiliza movableFaces
para generar los movimientos de estas capas.
--No se muestran detalles de este cálculo

-- | Devuelve las capas que pueden moverse
movableFaces :: BandagedCube -> [Face]
movableFaces bCube =
    difference [R, U, F, L, D, B] allBlockedFaces
    where
        allRestr = restrictions bCube
        allBlockedFaces = S.unions
            (S.map (facesBlockedByBlock) allRestr)

--Calcula las capas que bloquea un bloque (>=2 centros)
facesBlockedByBlock :: S.Set Int -> S.Set Face
facesBlockedByBlock xs
| (isBlockingCenters) = S.map (centerToLayer) centers
| otherwise = S.empty
where
    centers = S.filter (>= 48) xs
    isBlockingCenters = (length centers) >= 2

centerToLayer :: Int -> Face
centerToLayer 48 = U
centerToLayer 49 = F
centerToLayer 50 = R
centerToLayer 51 = B
centerToLayer 52 = L
```

Capítulo 4. Desarrollo e implementación

```
centerToLayer 53 = D
centerToLayer _ = N
```

4.5.2. Generación de heurísticas

Para generar las heurísticas, necesitamos almacenar el mínimo número de movimientos necesarios para resolver un subconjunto de piezas (esquinas y cada mitad de las aristas individualmente). Para ello, realizaremos una modificación del algoritmo de búsqueda en anchura, generando cada subconjunto de piezas por separado hasta una profundidad determinada. Por ahora, se van a generar para un cubo estándar, puesto que sus valores serán admisibles para cualquier configuración de bloqueos. No obstante, no se descarta generar heurísticas más precisas en el futuro para otras configuraciones. La generación nos aportará un vector de números enteros indexados estratégicamente según cada posible permutación y orientación de las piezas (ver 3.7.4). Inicialmente, se generan estas bases al inicio de la ejecución del algoritmo. No obstante, el objetivo será poder precomputarlas y almacenarlas en algún fichero, y de este modo solo será necesario cargarlas al inicio. De esta manera, podremos generar heurísticas a mayor profundidad, aunque su tiempo de generación sea mucho mayor.

Implementación

El algoritmo de búsqueda en profundidad suele implementarse con alguna cola de tipo FIFO (first in first out), donde al explorar cada nodo se almacenan sus vecinos para explorar después. Partimos desde el estado resuelto, a profundidad 0. Para cada estado, almacenaremos su profundidad p_0 en el vector de heurísticas, y generaremos todos los estados accesibles a un giro. Estos estados tendrán como profundidad un valor del intervalo $[p_0 - 1, p_0 + 1]$. Solo los casos de profundidad $p + 1$ serán nuevos, porque para los demás habremos encontrado una forma subóptima de generarlos. Por ello, es de crucial importancia generarlos en orden creciente de profundidad, y nunca empezar una capa de profundidad sin haber terminado la anterior (para esto nos sirve generarlos en anchura utilizando alguna cola FIFO). De no ser así, nuestras heurísticas no serían admisibles, por lo que hay que descartar estos estados que habíamos generado con antelación. Por último, una vez hayamos generado los estados a profundidad menor o igual que p , sabremos que $p + 1$ será una cota admisible para todos los estados a los que no hemos accedido.

Optimización

El número de estados accesibles a una profundidad determinada crece de forma exponencial (ver 3.7.1). Por ello, esta fase del algoritmo puede tener un tiempo de ejecución muy alto, y cualquier detalle que optimice su tiempo va a ser utilizado. Originalmente, se implementó con una FIFO clásica. Al analizar el desempeño del código, la operación de añadir elementos en ella consumía alrededor del 95 % del tiempo de ejecución, lo que era un claro cuello de botella. Esto se debe a que las inserciones se ejecutan en complejidad $\mathcal{O}(n)$. Por ello, se comenzaron a

utilizar colas con prioridad, donde la prioridad viene determinada por la profundidad de cada elemento. De esta manera, contamos con una cola análoga, pero sus inserciones se realizan en complejidad $\mathcal{O}(\log(n))$. Esto redujo significativamente su tiempo de ejecución, aunque no se ha medido con precisión debido a que fueron generados con códigos completamente diferentes. De esta manera, la operación de mayor tiempo se realiza en un 10% del mismo. Por otra parte, se utilizaron *Vector.Unboxed* para representar vectores de enteros. Estos se basan en posiciones de memoria contiguas en lugar de punteros, lo que es mucho más eficiente para tipos básicos. No obstante, las actualizaciones sobre estos seguían siendo extremadamente ineficientes. Esto se debe a la inmutabilidad de *Haskell*, puesto que realiza una copia con cada dato nuevo que se actualizaba. Por ello, se utilizaron vectores inmutables para registrar todos los cambios y convertirlos en vectores mutables. Para ello, se almacenaron todos los cambios a realizar, y posteriormente se actualizaban todos a la vez en los vectores de heurísticas. Solamente realizando este cambio, se obtuvo una ejecución alrededor de 350 veces más rápida. Dado que el valor máximo a almacenar es 20 [3], podremos utilizar enteros de un byte para gestionar la memoria más eficientemente. Esto consigue un consumo de memoria entre 2,5 y 5,5 veces menor. Finalmente, se puede optimizar el número de pasos del algoritmo. Cuando estemos recorriendo la capa de profundidad máxima, sabremos que no necesitamos explorar ninguno de sus adyacentes, dado que estarán repetidos o excederán nuestro umbral. Por ello, de no generarlos, ahorraremos la ejecución de una capa completa. Esta optimización logra una ejecución entre 8 y 10 más rápida, y un consumo de memoria alrededor de 9 veces inferior. Inicialmente, la generación de heurísticas de profundidad 3 (de 3.240 estados [3]) se ejecutaba en más de un minuto. Todas estas optimizaciones consiguen generar las heurísticas hasta profundidad 5 (con 574.908 estados [3]) en un tiempo razonable, inferior a 10 segundos.

4.6. Testing

Haskell cuenta con una librería para automatizar tests, llamada *quickCheck* [26]. Está basada en *Property Based Testing*, o testing basado en propiedades. En lugar de realizar pruebas unitarias manualmente, se especifican propiedades que deben de cumplir los valores. Posteriormente, se generan casos de prueba aleatorios y se comprueban si verifican lo especificado. En caso contrario, la librería devuelve un contraejemplo, que el programador puede analizar para corregir su código. Esto ayuda a encontrar los casos de error, que podrían no haberse contemplado en un primer momento. Por defecto se generan 100 casos de prueba, aunque pueden ser modificados. No haber encontrado un contraejemplo no garantiza que el código sea correcto, puesto que hay más factores que influyen en la corrección. Por ejemplo, pueden estar probándose características triviales, no probar todas las funciones del proyecto, no encontrar casos de error de muy baja probabilidad, o que los generadores aleatorios no cubran una cantidad significativa de los posibles casos. No obstante, es una herramienta muy útil para encontrar errores y darle más robustez al código. En este proyecto no se ha hecho un testing exhaustivo, pero se han probado las partes de código más importantes, críticas y susceptibles de contener errores difíciles de encon-

Capítulo 4. Desarrollo e implementación

trar. En primer lugar, se ha probado la codificación de los datos de los cubos y movimientos con las propiedades básicas de teoría de grupos:

- Asociatividad: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- Elemento neutro: $e \cdot x = x$ y $x \cdot e = x$
- Elemento inverso: $x \cdot x^{-1} = e$ y $x^{-1} \cdot x = e$
- Inversión doble: $(x^{-1})^{-1} = x$

En código *Haskell*, las propiedades quedan de la siguiente manera:

```
associativity :: Cube -> Cube -> Cube -> Property
associativity c1 c2 c3 = (c1 <> (c2 <> c3))
                         === ((c1 <> c2) <> c3)
```

```
neutral1 :: Cube -> Property
neutral1 c = (c <> mempty) === c
```

```
neutral2 :: Cube -> Property
neutral2 c = (mempty <> c) === c
```

```
inverse1 :: Cube -> Property
inverse1 c = (c <> (invert c)) === mempty
```

```
inverse2 :: Cube -> Property
inverse2 c = ((invert c) <> c) === mempty
```

```
inversedoble :: Cube -> Property
inversedoble c = ((invert . invert) c) === c
```

Las propiedades para algoritmos son análogas. Tras llamar a *quickCheck* con estas propiedades, el resultado es:

```
Testing Cube:
+++ OK, passed 100 tests.
```

```
Testing Moves:
+++ OK, passed 100 tests.
```

Test suite Bandaged-Cube-TFG-test: PASS

Para poder generar casos aleatorios, necesitamos implementar los generadores aleatorios de nuestros tipos. Esto se realiza instanciando la typeclass *Arbitrary*. En el caso del cubo, se generan listas aleatorias de permutaciones de los números [0, 53]. Evidentemente, muchas de ellas no serán estados posibles, pero lo que nos interesa es verificar si cumplen las propiedades de grupos. Sería mucho más interesante generar algoritmos aleatorios y aplicarlos al cubo, pero esto nos llevaría a dependencias circulares entre ambos ficheros. Para generar movimientos aleatorios, podemos almacenar las posibilidades en una lista (*possibleTurns*) y seleccionar uno aleatoriamente con la ayuda de la función *elements*. Finalmente, para generar algoritmos, podemos generar listas de movimientos aleatorias y simplificarlas.

```

instance Arbitrary Cube where
    arbitrary = do
        xs <- shuffle [0..53]
    return $ newCubeFromList xs

    -- | Makes a list with all the possible turns
    possibleTurns :: [Turn]
    possibleTurns = [Turn(turn, degrees) |
        turn <- [R ..], degrees <- [1..3]]

instance Arbitrary Turn where
    arbitrary = elements possibleTurns

instance Arbitrary Algorithm where
    arbitrary = do
        xs <- listOf arbitrary
    return (Algorithm xs <> mempty)
    --The <> mempty forces to simplify

```

Como se detalló en 3.7.4, las heurísticas han de ser admisibles. En caso contrario, el programa puede descartar constantemente los caminos y aumentar la profundidad, lo que lo lleva a no terminar la ejecución. Por ello, han sido objetivo del testing, y se han logrado encontrar errores previos en su implementación. En primer lugar, recordemos que las heurísticas se indexan basándose en la numeración factorial (S_8) y npr (${}_{12}P_6$). Estas numeraciones han de ser biyectivas, y podremos comprobar su inyectividad, es decir, que no hay dos elementos que reciben el mismo índice. Con esto, tendríamos una función de hashing perfecta. La sobreyectividad de esta función no útil de comprobar en este caso. Dado que siempre trabajamos con el mismo conjunto, esta propiedad no involucra la generación aleatoria de valores. Lo que haremos es generar todas las variaciones o permutaciones de estos conjuntos, asignarles un índice, ordenar la lista y comprobar que incluyen todos los valores sin duplicados. No se muestran detalles del cálculo de las variaciones y permutaciones. El código es el siguiente:

perfectHashingPerms :: Property

Capítulo 4. Desarrollo e implementación

```
perfectHashingPerms =
  property
    (sort numbering == result)
  where
    perms = permutations [0 .. 7]
    numbering = map (factorialNumbering) perms
    result = [minimum numbering .. maximum numbering]

perfectHashingNPR :: Property
perfectHashingNPR =
  property
    (sort numbering == result)
  where
    vars = variations 6 [0..11]
    numbering = map (nprNumbering [0..11]) vars
    result = [minimum numbering .. maximum numbering]
```

Posteriormente, podremos revisar si la heurística de Korf es admisible. Para ello, podemos generar un algoritmo aleatorio, realizar los giros pertinentes sobre un cubo sin bloqueos y comprobar si la heurística asigna un valor menor o igual al número de giros realizados.

```
korfAdmissible :: Algorithm -> Property
korfAdmissible alg = property (h <= lengthAlg alg)
  where
    finalSt = fromJust
      (tryToExecuteAlg newSolvedBandagedCube alg)
    h = korfHeuristic finalSt
```

Finalmente, obtendremos la propiedad más importantes del proyecto, la que prueba el propio solucionador. Esta genera un algoritmo aleatorio como mezcla, aplica estos movimientos a un cubo, calcula una solución y comprueba que al componer ambas secuencias se alcanza el estado resuelto. Además, también revisa que la longitud de la solución es menor o igual a la mezcla, dado que el algoritmo de Korf encuentra soluciones óptimas. Por motivos de eficiencia, se generan mezclas de tamaño máximo configurable. Esta propiedad también recibe como input los bloqueos del cubo y el generador aleatorio, puesto que algunos tienen un mal funcionamiento en algunos cubos con bloqueos. El output de esta propiedad ha sido modificado para que imprima las mezclas y soluciones halladas, además de terminar resumiendo la distribución de mezclas agrupadas por longitud de su solución óptima. Por último, se mide el tiempo para analizar su eficiencia.

```
korfSearchSolvesOptimally :: 
  Gen Algorithm          -- ^ Generador aleatorio
  -> [[Int]]             -- ^ Bloques
  -> Int                  -- ^ Tamano máximo
  -> Property
```

```

korfSearchSolvesOptimally
    customGenerator blocks maxLengthScramble =
let
    origin = newBandagedCube newSolvedCube blocks
in
forall (customGenerator)
    $ \scramble ->
let
    --ejecución de mezcla:
    scrambeledCube =
        fromMaybe origin
            (tryToExecuteAlg origin scramble)

    --solucionador:
    solve = fromJust
        (smartKorfSolver scrambeledCube)
    Algorithm xs1 = scramble
    Algorithm xs2 = solve
in

    --output:
    trace ("\\n\\nScramble: " ++ (show scramble) ++
        "\\nSolution: " ++ (show solve)) $

    --agrupación de datos
    collect (lengthAlg solve) $

    --Propiedad: resuelto con restricciones de longitud
    (solved (algToPerm (Algorithm (xs1 ++ xs2)))
        && (lengthAlg solve <= lengthAlg scramble)
    )

```

Esta propiedad devuelve un output del tipo:

Scramble: R2

Solution: R2

Scramble: L' B2 U2 L2 D2 F2

Solution: F2 D2 L2 U2 B2 L

Scramble: U D2 B' D2 B' U' R2

Solution: R2 U B D2 B U' D2

(...)

Testing Solutions:

TESTS OF SOLUTIONS AND SEARCH ENGINE

Testing a 3x3x3

Capítulo 4. Desarrollo e implementación

```
+++ OK, passed 100 tests:
```

```
13% 1
13% 8
10% 6
9% 0
9% 10
9% 3
9% 5
9% 7
7% 4
6% 2
6% 9
```

```
real    1m39,328s
user    1m38,417s
sys     0m1,053s
```

Este output contiene mezclas para un 3x3 tradicional de un máximo de 10 movimientos. En este caso, las mezclas podrían generarse mediante giros aleatorios. No obstante, se mejoró el generador para generar secuencias canónicas y descartar algunos movimientos en cada momento. Esta idea funciona muy mal para cubos con bloqueos, en especial aquellos con muchas restricciones y algunas capas restringidas por completo. Debido a que algunos movimientos no son válidos por romper bloques, se descartan y reducen la longitud del algoritmo. Tras este descarte, los movimientos adyacentes a él tienen más probabilidad de ser de la misma capa y cancelarse, dando lugar a otra reducción de la longitud y más opciones de volver a cancelar los movimientos anteriores y posteriores. Por ello, este generador lograba mezclas muy cortas, normalmente resolubles en menos de 5 giros. Se probó la generación en función de las capas útiles en cada momento, pero su generación fue altamente ineficiente. Por ello, se optó por almacenar algunas mezclas manualmente con una longitud considerable, y generar subsecuencias aleatorias de estas. Aunque la solución es poco elegante y se prueban casos de un dominio muy reducido, nos sirve para comprobar que el solucionador resuelve cubos con configuraciones de bloques diferentes. Tras el testing de estas propiedades, podemos asegurar un funcionamiento correcto del código.

Capítulo 5

Resultados experimentales

En este momento, se cuenta con un solucionador general para cubos de Rubik con bloqueos. Permite introducir una situación arbitraria, encontrar una solución óptima mediante el algoritmo de Korf, y generar un vídeo resolviendo el cubo. También define una librería de funciones, lo que permite utilizar el software de otras maneras. El solucionador funciona en un tiempo razonable, en especial para cubos con muchas restricciones. Para cubos con pocas o ninguna restricción, su tiempo en las secuencias más alejadas del estado resuelto puede ser del orden de horas. Este orden de tiempos es normal en este tipo de problemas [3] [11]. En ningún caso ha agotado la memoria del ordenador. La solución que devuelve es correcta para el 100 % de los casos de un input correcto, o por lo menos no se ha encontrado ningún contraejemplo donde la solución no resuelva el cubo o rompa algún bloque.

5.1. Evaluación del rendimiento

Para analizar el rendimiento del código, se han generado 100 casos de prueba aleatorios para cada cubo utilizando *quickCheck*, y se ha medido el tiempo de ejecución total. Con esto, podemos estimar el tiempo promedio de cada resolución, puesto que se ha tratado que los tests incluyan resoluciones de tamaños diversos. Cabe destacar que en este tiempo se incluye la generación aleatoria de mezclas, la generación de heurísticas y la resolución del cubo. Se han utilizado el cubo estándar, varios subgrupos del cubo tradicional y los cubos *alcatraz*, *bicube* y el mostrado por *TheMaoiSha* [2]. A este cubo le llamaremos *TheMaoiSha-252*. Todos los casos se han compilado con la optimización -O2, y han utilizado heurísticas de profundidad 5. Para los subgrupos del cubo, se ha acotado la profundidad por motivos de eficiencia. A excepción del subgrupo $\langle R, U \rangle$, algunas situaciones largas pueden llevar horas en resolverse. Esto es debido a la naturaleza exponencial de nuestro problema. Como vimos en 4.6, a la hora de generar casos de prueba en el solucionador, podemos agrupar y contar las longitudes de las soluciones óptimas, lo que nos permite calcular su media.

En la tabla 5.1 se muestra un resumen de los casos de prueba generados para medir la eficiencia del solucionador. Los resultados del tiempo total y promedio

Capítulo 5. Resultados experimentales

por cada cubo se muestran en la tabla 5.2.

Cubo	Profundidad máxima	Profundidad media
3x3	10	4.96
$\langle R, U \rangle$	20	10.31
$\langle R, U, L \rangle$	13	6.52
$\langle R, U, F \rangle$	12	6.59
$\langle R, U, F, L \rangle$	11	5.48
Alcatraz	22	9.85
Bicube	22	9.19
TheMaoiSha-252	252	72.08

Cuadro 5.1: Resumen de las mezclas generadas

Cubo	Tiempo total	Tiempo medio por cubo
3x3	1 min 40 s	1 s
$\langle R, U \rangle$	9 min	5.4 s
$\langle R, U, L \rangle$	2 min	1.2 s
$\langle R, U, F \rangle$	2 min	1.2 s
$\langle R, U, F, L \rangle$	1 min 45 s	1.05 s
Alcatraz	50 s	0.5 s
Bicube	45 s	0.45 s
TheMaoiSha-252	3 min 45 s	2.25 s

Cuadro 5.2: Evaluación del rendimiento

En los primeros casos, se ajustó la profundidad máxima para que los resultados fueran similares, a excepción del subgrupo $\langle R, U \rangle$. En este, se generaron mezclas de como mucho su profundidad máxima, 20 [27], y tardó 9 minutos en generar y resolver las mezclas. Tras anotar las mezclas generadas en un fichero y resolverlas directamente, tardó 1 minuto, lo que nos da a entender que la mayoría del tiempo se consume generando mezclas. No obstante, la generación de los cubos con bloqueos irregulares no contó con estas podas, lo genera un análisis más fiel a la realidad.

Como comentamos, por ahora se generan las bases de datos al inicio, con idea de un futuro poder guardarlas en ficheros y cargarlas en cada ejecución. Este proceso suele tardar entre 5 y 10 segundos. Dado que estas bases son siempre las mismas, en los casos de test muy seguramente hayan sido generadas una única vez. Teniendo esto en cuenta, podemos comprobar que el tiempo de la resolución es bajo, o incluso despreciable en comparación con la generación de las bases de datos.

Por último, se ha estudiado con algo más de detalle la ejecución de la resolución de 252 movimientos. Se pueden generar varios *profilings* con los comandos:

```
#Garbage collector
cabal run exe:Bandaged-Cube-TFG -- +RTS -s
```

5.1. Evaluación del rendimiento

```
#Distribución de cada función
cabal run exe:Bandaged-Cube-TFG -- +RTS -p

#Memoria en heap
cabal run exe:Bandaged-Cube-TFG -- +RTS -hm
```

El análisis del *garbage collector* incluye mucha información, en especial sobre el rendimiento en tiempo y memoria. Un resumen del resultado del profiling es el siguiente:

```
73,031,260,192 bytes allocated in the heap
252,536,624 bytes maximum residency (17 samples)
 605 MiB total memory in use
    (0 MiB lost due to fragmentation)
```

INIT	time	0.000s	(0.000s elapsed)
MUT	time	18.087s	(18.065s elapsed)
GC	time	1.643s	(1.647s elapsed)
EXIT	time	0.000s	(0.000s elapsed)
Total	time	19.730s	(19.712s elapsed)

Productivity 91.7% of total user, 91.6% of total elapsed

Primero, muestra el coste total de asignaciones de memoria (73 GB), la memoria total utilizada (605 MiB) y la memoria máxima que se mantuvo en memoria sin ser recolectada por el *garbage collector* (252 MB). Evidentemente, este valor es mayor que los 173 MB estimados en 3.7.4 para almacenar las bases de datos. Posteriormente, se divide la ejecución en sus fases. Son de interés MUT (la ejecución de nuestro código) y GC (*garbage collector*), puesto que las demás son prácticamente 0. La eficiencia es el porcentaje de la ejecución de MUT con respecto al total. Observamos que es superior al 90%, lo que nos da a entender que no realizamos asignaciones de memoria de poca utilidad que supongan una ralentización significativa de nuestro código.

Posteriormente, podemos realizar otro profiling para analizar el tiempo y memoria consumida por cada función. Se muestran las 5 funciones que más tiempo consumen, además del resto de funciones propias.

COST CENTRE	MODULE	%time	%alloc
balanceR	Data.Set.Internal	10.2	23.2
slicePieces	Cube	7.2	13.5
turnPreserveBlock.s2Real	Bandaged	5.8	3.4
toAsclList1	Data.Set.Internal	5.2	7.7
link	Data.Set.Internal	5.0	4.1
(...)			
turnPreserveBlock.s1Real	Bandaged	3.5	2.1
piecesNotAffected	Bandaged	3.2	0.1
edgesKey.	IndexHeuristics	1.2	2.4

Capítulo 5. Resultados experimentales

<>	<i>Cube</i>	1.0	0.6
<i>edgesState.xs</i>	<i>MathNotation</i>	0.9	2.9
<i>edgesKey.allp</i>	<i>IndexHeuristics</i>	0.7	1.8
<i>cornerState.xs</i>	<i>MathNotation</i>	0.6	1.6

Como podemos observar, la función que más tiempo acumula es *balanceR*, de la biblioteca de conjuntos *Data.Set*. Debido a que este tiempo es bajo (del orden del 10 %), podemos concluir que no hay ningún cuello de botella significativo en lo que a tiempo de ejecución se refiere. En caso de querer optimizar el código, se puede comenzar con las funciones que más recursos consumen, o también se puede tratar de reducir su número de cálculos necesarios.

Por último, se muestra un diagrama del consumo de memoria en heap de algunos módulos (figura 5.1). Cabe destacar que no ha de tomarse como referencia el eje temporal, puesto que la información requerida en estos análisis ralentiza la ejecución. En la figura, se observan 3 picos al inicio, los cuales representan la generación de heurísticas de cada uno de los 3 tipos de piezas. Al final de cada uno de ellos, hay una liberación de memoria del *garbage collector*. Posteriormente, y durante la fase de resolución, su consumo es constante, puesto que se mantienen en memoria los vectores de heurísticas. Además, este valor ronda los 175 MB, lo cual cuadra con lo calculado en 3.7.4. Curiosamente, no se muestra el consumo del módulo *Search*. Esto se debe a que su uso de memoria es despreciable.

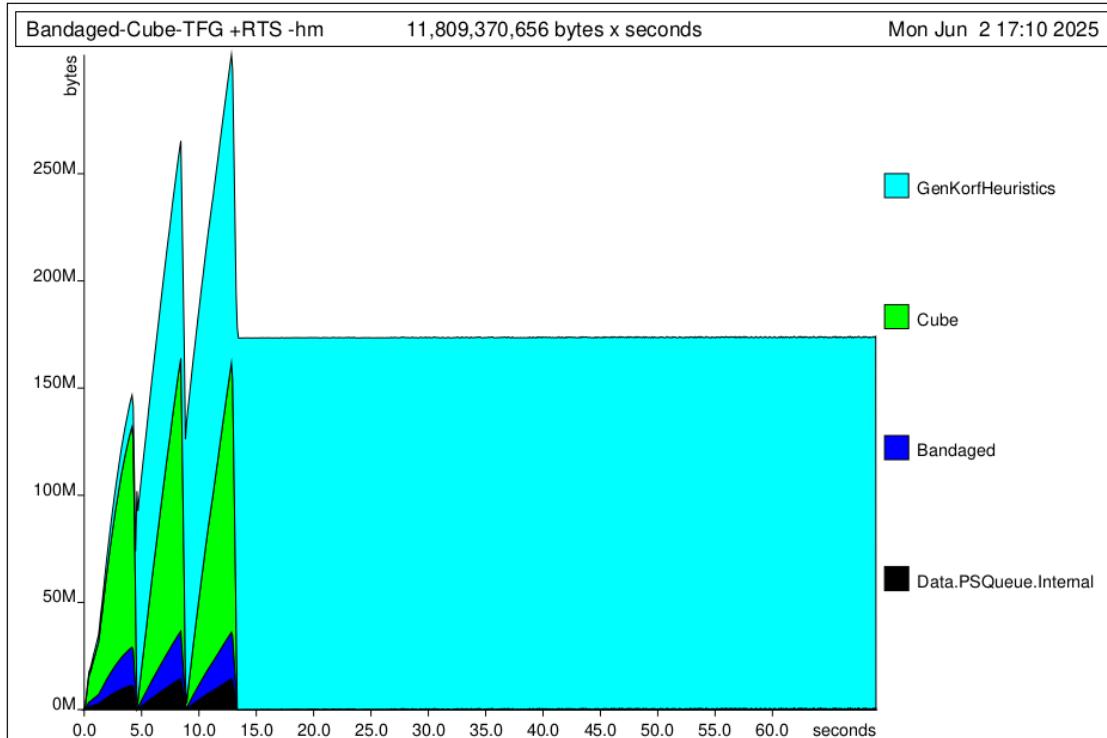


Figura 5.1: Consumo de memoria de una resolución de TheMaoiSha252

Esto podemos corroborarlo eliminando las heurísticas, y realizando la misma

5.2. Contribuciones teóricas y demostraciones

medición en un algoritmo ciego (en este caso, IDDFS, de una mezcla mucho más corta). Esto se muestra en la figura 5.2, en la que podemos observar que el módulo *Search*, al igual que otros como *Cube* o *Bandaged* tan solo necesitan unos pocos kilobytes de memoria.

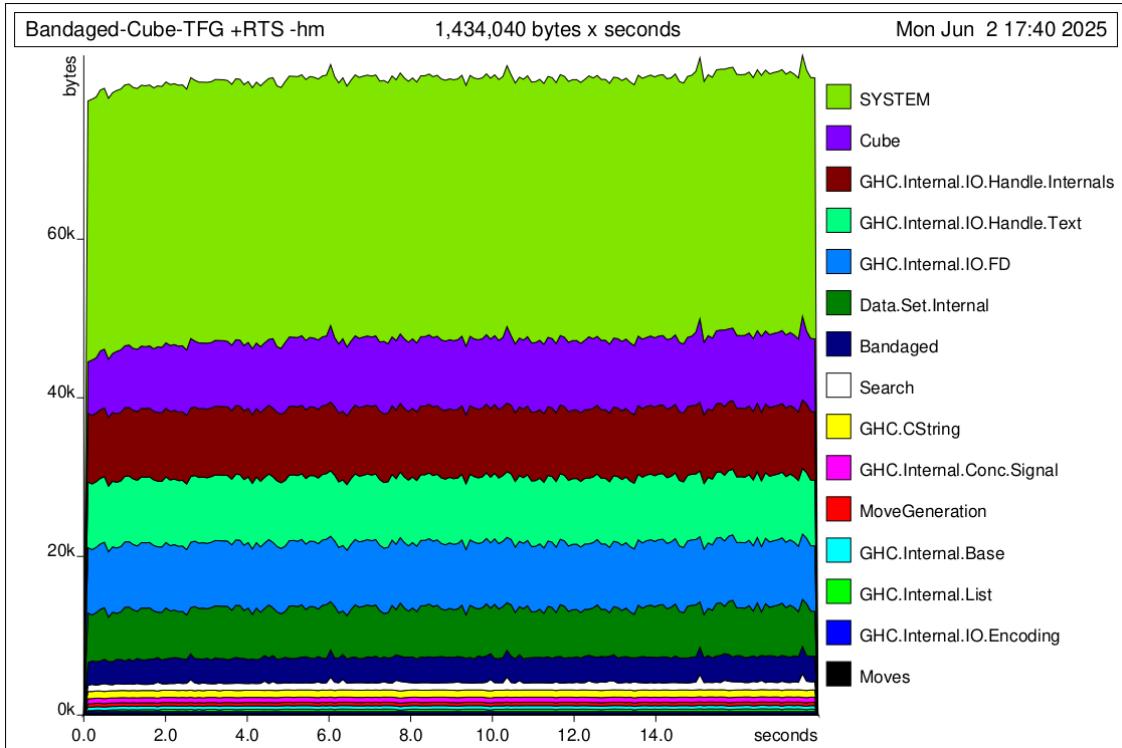


Figura 5.2: Consumo mediante IDDFS

5.2. Contribuciones teóricas y demostraciones

Una pregunta que surje en este tipo de cubos es si su número máximo necesario para resolverlo es 20 como en el 3x3 convencional [3], o si hay configuraciones que permiten secuencias más largas. Aunque ya se conocen algunos cubos con secuencias óptimas de más de 20 movimientos [4], no existe mucha información al respecto. Podemos responder que 20 no es una cota correcta, puesto que he hallado configuraciones para los cubos *alcatraz* y *bicube* para las que el programa encuentra una solución óptima de 22 movimientos. Para estos estados, el código tarda en encontrarlas aproximadamente 13 y 8,5 segundos respectivamente. Cabe destacar que se han generado con un programa con un input y output distinto, pero el algoritmo es idéntico. Esto nos indica que para estos cubos, la cota mínima de su *número de Dios* es 22, y nada nos garantiza que no sea un número mayor.

Para el Bicube:

Scramble of length 22: *F' U L F' L' F2 R2 U2 L' U R' U2 L U' F' U' F R U L' U2 R*

Capítulo 5. Resultados experimentales

*Solution of length 22 found: R' U2 L U' R' F' U
F U L' U2 R U' L U2 R2 F2 L F L' U' F*

Para el cubo Alcatraz:

Scramble: of length 22: U F' U' F2 R' F' R U F

U' R' F' U' R U F R' F' R2 U' R' U

Solution of length 22 found: U' R U R2 F R F'

U' R' U F R U F' U' R' F R F2 U F U'

Por otra parte, el youtuber *TheMaoiSha* mencionó en un vídeo [2] que halló una secuencia de 252 movimientos para una situación de un cubo, y comentó que no había secuencias más cortas para resolver ese estado. Tras probar esta secuencia en el programa, también encontró una secuencia de 252 movimientos para ese estado en unos 20 segundos aproximadamente. El código halla su secuencia inversa (solo voltear 4 aristas sin permutar más piezas, por lo que es autoinversa), pero ambas secuencias son equivalentes y realizan el mismo efecto en el cubo. Esto corrobora su afirmación, puesto que la secuencia que encontró es óptima.

Solution of length 252 found:

R U F R2 F' R' U F2 U' F' R U' F' U2 F R'

U' R2 U F R U F R U2 R' U' F R2 F' R' U F'

R' F2 R U' F' U2 F R U F R U F2 U' F' R U2

R' U' F R' U' R2 U F' R' F2 R U F

(repetido 4 veces)

Estos son solo algunos de los descubrimientos y aportes que he conseguido realizar con este software. Este nos puede ayudar a encontrar nuevos resultados de cubos con bloqueos concretos, sobre los cuales existen pocos trabajos. Otra utilidad puede ser encontrar o acotar el número mínimo necesario para resolver algunos subgrupos del cubo, puesto que en ocasiones pueden obtenerse mediante algunos bloqueos concretos. Existen algunos resultados ya descubiertos [27], aunque todavía hay algunos subgrupos cuyo número mínimo no ha sido calculado.

Capítulo 6

Análisis de impacto

Esta herramienta puede tener un impacto en varios campos. En primer lugar, puede ser de gran utilidad en el ámbito académico, puesto que los cubos de Rubik han sido un campo de estudio con multitud de publicaciones, en especial en los campos de las matemáticas, informática e inteligencia artificial [11], [3], [14]. Esta puede ser utilizada para realizar descubrimientos sobre los cubos con bloqueos, como por ejemplo el número máximo de movimientos necesarios para resolver algunas configuraciones concretas.

Por otra parte, cuenta con posibles aplicaciones en el ámbito tecnológico. Esta herramienta materializa una posible solución de un problema real. También puede ilustrar el uso de algunos principios matemáticos en la informática, reflejando su histórica interdependencia entre ambas disciplinas. Puede ser útil analizar sus características y compararlas con las de otras propuestas similares. Por ejemplo, podrían reflejar las limitaciones y ventajas de sus herramientas (como *Haskell*) o de sus métodos (como el algoritmo de Korf o resultados de la teoría de grupos) con otras posibles soluciones (por ejemplo mediante el uso del *machine learning* u otros lenguajes de programación). Además, algunas de las técnicas, algoritmos y optimizaciones de código pueden ser extrapoladas para otros cubos para crear nuevos solucionadores, mejorar los que existen o incluso utilizarse en otros problemas diferentes.

Otro contexto donde puede ser de utilidad es el ámbito educativo. Puede fomentar la práctica de resolución de cubos de Rubik, un pasatiempo con multitud de ventajas. También anima a tratar de resolver aquellos con bloqueos, los cuales pueden ser muy diferentes de resolver. Este trabajo puede fomentar el pensamiento lógico-matemático y la resolución de problemas, además de los conocimientos sobre la programación, algorítmica o matemáticas. El código de este software será publicado, y gracias a ser de *open-source*, permitirá democratizar su uso.

También puede ser de utilidad en el ámbito cultural y lúdico. El cubo de Rubik se ha consolidado como un elemento ampliamente reconocido. Cada avance, publicación en medios, herramienta o creación ayuda a fomentar y preservar este ícono, y este software puede ser una de ellas. Por otra parte, también puede

Capítulo 6. Análisis de impacto

influir en el ámbito social. El cubo es un punto de encuentro entre aficionados, competidores, coleccionistas o modificadores de todo el mundo, y este software puede mejorar la cohesión entre diferentes usuarios de todo el mundo. Dada la naturaleza autodidacta y *open-source* de esta comunidad, permitiría, por ejemplo, que los usuarios pudieran generar algunos algoritmos de estos cubos o desarrollar nuevos métodos de resolución. Esto puede fomentar que publiquen sus descubrimientos en cualquier plataforma y mejore nuestro conocimiento sobre estos cubos.

En cuanto al impacto personal de este proyecto, ha supuesto un aprendizaje inmenso. He podido aplicar multitud de conocimientos, tanto en mis estudios universitarios como los aprendidos de manera autodidacta. Me ha permitido observar cómo pueden complementarse diferentes ramas de conocimiento aparentemente desconexas. Me ha servido como excusa para aprender sobre algunos temas, como la teoría de grupos, algoritmos o la programación funcional. Por otra parte, dada la novedad de este proyecto, me ha obligado a idear soluciones nuevas que nadie ha realizado, además de investigar sobre trabajos similares y saber detectar qué ideas pueden ser adaptadas a este problema. También me ha permitido gestionar un proyecto relativamente grande, de más de 1500 líneas de código en *Haskell* y otros lenguajes. Me ha hecho hecho gestionar la integración de herramientas en un mismo proyecto, incluso estando escritas en diferentes lenguajes. La magnitud del proyecto te hace ser consciente sobre algunos aspectos en la gestión del código, el diseño en módulos de la manera más independiente posible, la ingeniería de software, el testing y similares.

No obstante, parte del impacto de este proyecto puede ser negativo. Uno de sus efectos adversos puede ser que esta herramienta desincentive a resolver los retos por uno mismo. Como te aporta una solución para cualquier estado, puede desvirtuar el desafío y fomentar la cultura del resultado en lugar de la del procedimiento seguido. Aun así, este efecto será logrado o evitado en función de cómo se use esta herramienta.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

En este trabajo, se ha conseguido realizar un solucionador general de cubos de Rubik de con bloqueos. Este programa cuenta con una interfaz para su uso por un usuario no experimentado, y muestra una visualización de la solución. Resuelve el puzzle correctamente y de forma óptima en un tiempo razonable, el cual disminuye a medida que aumentan las restricciones. Para cubos *bandaged* convencionales, su tiempo de resolución no excede del medio minuto. Para cubos con pocos o ningún bloqueo, este tiempo puede ser del orden de horas, aunque otros solucionadores han tenido tiempos similares para problemas como este [11]. No obstante, están previstas algunas mejoras para conseguir reducir este tiempo en todos los casos. Gracias a esta herramienta, se han encontrado soluciones de algunos bloqueos cuyo número de movimientos necesarios excede al del 3x3 convencional [3]. Más concretamente, se han encontrado la solución de secuencias de 22 movimientos de los *alcatraz* y *bicube*, y una solución de 252 para un cubo con más bloqueos. Esta herramienta podrá ser utilizada para descubrir algunas propiedades y resultados de estos cubos.

7.2. Trabajo futuro

Optimización y mejoras de eficiencia El primer punto que puede mejorarse de este proyecto es el rendimiento del solucionador. Aunque funciona razonablemente bien dentro de su propósito, el de los cubos con muchos bloqueos, siempre hay margen de mejora, y el tiempo en hallar una solución puede ser reducido. Esto será de mayor importancia para los cubos con pocos bloqueos dada la extensión de su árbol de búsqueda. Se pueden almacenar las bases de datos de heurísticas en ficheros y cargarlas al comienzo de cada ejecución. De esta manera, eliminaremos los 5-10 segundos de generación a profundidad 5, y sumaremos el tiempo de lectura de los ficheros. Esto nos permitirá precomputar heurísticas de mayor profundidad, con una dependencia menor de este tiempo. Otra alternativa que puede probarse consiste en utilizar heurísticas basadas en las posiciones de los bloques. Habría que analizar en detalle qué opciones

Capítulo 7. Conclusiones y trabajo futuro

son viables y tienen un tamaño razonable, pero podría mejorar el rendimiento del solucionador. También podrían generarse heurísticas para caso de cubo con bloqueos, y tener una cota admisible más ajustada. No obstante, podría ser difícil de generalizar, y habría que considerar que el usuario tenga una configuración de bloqueos idéntica desde otra posición. La eficiencia también puede intentar ser mejorada mediante el paralelismo, probando algoritmos concurrentes, distribuidos o que resuelvan varios cubos simultáneamente.

Mejoras en el input, output y corrección de errores La parte más débil de este proyecto es el input de una posición. Sería interesante contar con una interfaz más sencilla de usar, y mejorar la detección de errores con la que cuenta. Podría diseñarse una interfaz que permita observar el input según se introduce, detectar y corregir errores. Un ejemplo de programa con una buena interfaz es *cubexplorer* [1], de Herbert Kociemba. Por otra parte, también puede ser interesante mejorar el output y dar más opciones. Por ejemplo, en lugar de generar un vídeo, permitir que el usuario confirme al programa cada giro que hace, y no se realicen a velocidad constante. También puede ser interesante implementar opciones de accesibilidad, por ejemplo leyendo en voz alta el algoritmo, o incluyendo la información de las piezas no solo con colores (véase con diagramas o formas).

Nuevas funcionalidades Este software se ha diseñado de forma modular, con el objetivo de facilitar la integración de nuevas funcionalidades. Por ejemplo, podrían probarse métodos de resolución en varios pasos que, aunque su solución no sea óptima, puedan tener un tiempo reducido. Un ejemplo muy conocido es el método de 2 fases de Herbert Kociemba [19], y puede ser buena idea buscar métodos basados en la misma idea. De forma similar, también puede explorarse la búsqueda de algoritmos subóptimos. Estos priorizan el tiempo de ejecución por encima de la longitud de la solución. Otra idea interesante de implementar sería poder generar todos los estados de un cubo en orden de cercanía a la solución creciente. Con ello, podríamos calcular el diámetro de su grafo. Este puede ser un reto complejo debido a la alta cantidad de permutaciones posibles de algunos cubos. También puede ser interesante extender todas las ideas de este trabajo a otros cubos similares, como versiones con bloqueos del 2x2x2 o como el *Square-1*. Este último puede considerarse como una versión bloqueada del *Square-2*, el cual es un grupo. Estas ideas pueden ser aplicadas a cubos no necesariamente con bloqueos. Por ejemplo, podría adaptarse el algoritmo de Korf a otros cubos, en los que los métodos simples (como búsqueda en anchura) no funcionen en un tiempo razonable. Algunos ejemplos de puzzles de permutaciones con un número de estados relativamente similar pueden ser el *Face-Turning Octahedron* (de $3,14 \cdot 10^{22}$ estados), el *Rubik's Clock* (de $1,28 \cdot 10^{15}$ posibilidades) o el cubo helicóptero (de $4,94 \cdot 10^{17}$ alternativas). También puede ser interesante extrapolar algunas técnicas a cubos con órdenes mayores, como otros $n \times n \times n$ con o sin bloqueos, u otros puzzles como el *Megaminx*. No obstante, debido a su alta cantidad de nodos, es posible que hallar soluciones óptimas sea un problema intratable, y sea necesario diseñar buscadores de varias fases. Otra cuestión que puede mejorarse es el testing, en especial el que comprueba propiedades de teo-

ría de grupos. Sería muy interesante mejorarlo y, además de aportar robustez al código, permitiría comprobar de forma experimental algunos resultados del cubo. Esto puede tener un interés académico y educativo, y puede extrapolarse a otros puzzles.

Aplicaciones y utilidades Finalmente, lo más interesante que se puede hacer con este proyecto es utilizarlo para encontrar resultados. Puede ser útil para generar algoritmos y poder crear métodos de resolución completos. También puede ser una herramienta muy interesante calcular o acotar el máximo número de giros necesario para algunas configuraciones de bloqueos. De esta forma, sacaríamos el máximo partido posible de este software, y puede ser crucial para aumentar nuestro conocimiento sobre los cubos con bloqueos. La creación de esta herramienta puede ser solo el comienzo.

Bibliografía

- [1] «Cube Explorer». (2018), dirección: <https://kociemba.org/cube.htm>.
- [2] «Este CUBO de Rubik es DEMENCIAL | Récord a la secuencia más larga!» (2023), dirección: <https://www.youtube.com/watch?v=gqBemWZscrk&t=499s>.
- [3] M. D. Tomas Rokicki Herbert Kociemba y J. Dethridge, «The Diameter of the Rubik's Cube is Twenty», *SIAM Journal on Discrete Mathematics*, 2013.
- [4] «BiCube / Bandaged Rubik's Cube». (2009), dirección: <https://www.jaapsch.net/puzzles/bandage.htm>.
- [5] ladislavdubravsky, 2017. dirección: <https://github.com/ladislavdubravsky/bandaged` .2cube` .2explorer>.
- [6] «TheMaoiSha». (2011-actualidad), dirección: <https://www.youtube.com/@TheMaoiSha>.
- [7] «10 HORAS DE SUFRIMIENTO!» (2023), dirección: <https://www.youtube.com/watch?v=6gD4IIStIjw>.
- [8] paulnasca, *paul_cube_bandaged_graph*, 2019. dirección: https://github.com/paulnasca/paul_cube_bandaged_graph.
- [9] JoshMermel, *bandaged-cube-explorer*, 2020. dirección: <https://github.com/JoshMermel/bandaged` .2cube` .2explorer>.
- [10] JackTriton, *bandaged-cube-explorer*, 2023. dirección: <https://github.com/JackTriton/bandaged` .2cube` .2explorer>.
- [11] R. E. Korf, «Finding Optimal Solutions to Rubik's Cube Using Pattern Databases», 1997.
- [12] R. E. Korf, «Depth-First Iterative-Deepening: An Optimal Admissible Tree Search», *Artificial Intelligence*, 1985.
- [13] J. Mulholland, *Permutation puzzles*. Self Published, 2011.
- [14] D. Singmaster, *Notes on Rubik's Magic Cube*. Enslow Publishers, 1981.
- [15] «Package example using the rubikcube package». (2014), dirección: <https://www.overleaf.com/latex/examples/package` .2example` .2using` .2the` .2rubikcube` .2package/vddtnnnrdzrz>.
- [16] «List of method». (2012), dirección: https://www.speedsolving.com/wiki/index.php>List_of_methods.

BIBLIOGRAFÍA

- [17] «Some Rubik's Cube codes». (2024), dirección: <https://github.com/Alexsql/Rubik>.
- [18] «52-move algorithm for Rubik's Cube». (1981), dirección: <https://www.jaapsch.net/puzzles/thistle.htm>.
- [19] «The Two-Phase Algorithm Coordinates». (1992), dirección: <https://kociemba.org/math/twophase.htm>.
- [20] «Manim Rubik's Cube». (2021), dirección: <https://manim.2rubikscube.readthedocs.io/en/stable/>.
- [21] *A library for boxed vectors*. 2008. dirección: <https://hackage.haskell.org/package/vector`20.13.2.0/docs/Data`2Vector.html>.
- [22] *A library for sets, Data.Set*, 2002. dirección: <https://hackage.haskell.org/package/containers`20.8/docs/Data`2Set.html>.
- [23] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed., p. 209): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [24] R. E. Korf y P. Schultze, «Large-scale parallel breadth-first search», en *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, 2005.
- [25] N. T. van Doorn, *A group is a monoid with invertibility*, 2013. dirección: <https://hackage.haskell.org/package/groups`20.5.3/docs/Data`2Group.html>.
- [26] N. S. Koen Claessen Björn Bringert, *QuickCheck: Automatic testing of Haskell programs*, 2000. dirección: <https://hackage.haskell.org/package/QuickCheck`22.15.0.1/docs/Test`2QuickCheck.html>.
- [27] «God's Algorithm». (2024), dirección: https://www.speedsolving.com/wiki/index.php?title=God%27s_Algorithm.

Anexos

Apéndice A

Cálculo de los factores de ramificación de algunos subgrupos

Como se detalló en el capítulo 3, algunos bloqueos pueden impedir el giro de capas completas. Esto nos permite reducir el árbol de búsqueda sobre el que buscamos una solución. Se va a detallar el cálculo del factor de ramificación de algunos subgrupos. Una vez tenemos este número (llamémoslo b), podemos aproximar el espacio de búsqueda de n giros a b^n . Para ello, utilizaremos las secuencias canónicas. Estas son aquellas que nunca repiten el mismo movimiento y que nunca duplican las generaciones de las capas opuestas. Por ejemplo, las secuencias RL y LR efectúan el mismo efecto en el cubo, por lo que solo serán generadas una vez. En este anexo, solo nos centramos en subgrupos del cubo que bloquean o liberan capas por completo. Existen más subgrupos, como los que solo permiten giros dobles en las capas, los cuales no se utilizan en este proyecto. Algunos subgrupos no pueden crearse a partir de bloqueos, por ejemplo el subgrupo $\langle R, L, U, D \rangle$, por lo que no lo detallaremos. Utilizamos la notación de subgrupo generador, que equivale a todos los estados que podemos generar moviendo algunas capas. Por ejemplo:

$$\langle R, U, F \rangle = \{R, RU, R'U2, FU2R, RURUF'R2 \dots\}$$

Aunque este conjunto contiene una cantidad de secuencias infinita, las permutaciones accesibles desde un cubo resuelto aplicando estos movimientos será finita.

A.1. Dos capas contiguas, $\langle R, U \rangle$

A la hora de seleccionar el primer movimiento de una secuencia, tenemos $3 \cdot 2 = 6$ opciones, puesto que hay 2 capas y 3 opciones de giro en cada una. Para el resto de giros, solo tenemos una capa que mover para no repetir la capa anterior, lo que nos da lugar a 3 alternativas. Por ello, para n giros, tendremos $6 \cdot 3^{n-1}$

Capítulo A. Cálculo de los factores de ramificación de algunos subgrupos

secuencias canónicas. Esto nos da un factor de ramificación promedio de 3, como se muestra en la tabla A.1.

n giros	q_n secuencias canónicas	Factor de ramificación ($\frac{q_n}{q_{n-1}}$)
1	6	-
2	18	3
3	54	3
4	162	3
5	486	3
6	1458	3
7	4374	3
8	13122	3
9	39366	3
10	118098	3
:	:	:

Cuadro A.1: Cálculos para 2 capas

A.2. Tres capas que comparten una esquina, $\langle R, F, U \rangle$

Este cálculo es similar al de 2 capas contiguas. Para el primer movimiento, podemos escoger entre $3 \cdot 3 = 9$ opciones, 3 por cada una de las 3 capas. Para el resto de giros, tendremos $3 \cdot 2 = 6$ alternativas. Por tanto, para n giros, tendremos $9 \cdot 6^{n-1}$ secuencias canónicas. Esto nos da un factor de ramificación promedio de 6, como se muestra en la tabla A.2.

n giros	q_n secuencias canónicas	Factor de ramificación ($\frac{q_n}{q_{n-1}}$)
1	9	-
2	54	6
3	324	6
4	1944	6
5	11664	6
6	69984	6
7	419904	6
8	2519424	6
9	15116544	6
10	90699264	6
:	:	:

Cuadro A.2: Cálculos para 3 capas contiguas

A.3. Tres capas, de las cuales 2 opuestas, $\langle R, L, U \rangle$

Este cálculo no es tan sencillo como el anterior, puesto que se basa en una relación de recurrencia. Por simplicidad, vamos a comenzar calculando las cadenas

A.3. Tres capas, de las cuales 2 opuestas, $\langle R, L, U \rangle$

de n símbolos del conjunto $\{R, U, L\}$ (es decir, solo las capas seleccionadas), y al final multiplicaremos por 3^n para hallar las secuencias canónicas. Vamos a definir 2 conjuntos: r_n serán las secuencias de n capas que comienzan por R , y s_n las que no empiezan por R . Es fácil ver que estos conjuntos son disjuntos, y $q_n = 3^n(s_n + r_n)$. Vamos a generar algunos casos pequeños, y veremos el patrón posteriormente:

$s_1 :$		r_1
U		R
L		
<hr/>		
$s_2 :$		r_2
UR		RU
LU		RL
UL		
<hr/>		
$s_3 :$		r_3
URU		RUR
URL		RUL
LUR		RLU
LUL		
ULU		
\vdots		\vdots

Para generar nuevos casos de r_n , podemos tomar los de s_{n-1} y añadir una R al inicio. Siempre será una secuencia canónica, puesto que R aparece antes de L o U . Para generar los de s_n , tomamos los de r_{n-1} y le añadimos una U al inicio, y tomamos los de s_{n-1} y añadimos la letra contraria a la que inicie la cadena (una L si empieza por U y viceversa). Por tanto, las sucesiones tienen la siguiente forma:

$$\begin{cases} s_n = s_{n-1} + r_{n-1} \\ r_n = s_{n-1} \\ s_1 = 2 \\ r_1 = 1 \end{cases}$$

Operando, tenemos que:

$$s_n = s_{n-1} + r_{n-1} = s_{n-1} + s_{n-2}$$

Esta sucesión es similar a la sucesión de Fibonacci (f_n), aunque con otras condiciones iniciales. Como $s_1 = 2 = f_3$ y $s_2 = 3 = f_4$, tenemos que

$$s_n = f_{n+2}$$

Capítulo A. Cálculo de los factores de ramificación de algunos subgrupos

Finalmente, el número de secuencias canónicas es

$$q_n = 3^n(r_n + s_n) = 3^n(s_n + s_{n-1}) = 3^n s_n = 3^n \cdot f_{n+3}$$

Esta sucesión es equivalente a

$$\begin{cases} q_n = 3 \cdot s_{q-1} + 9 \cdot q_{n-2} \\ q_1 = 9 \\ q_2 = 45 \end{cases}$$

Calculando algunos valores, obtenemos un factor de 4.85, como se aprecia en la tabla A.3.

n giros	f_{n+3}	q_n secuencias canónicas	Factor de ramificación ($\frac{q_n}{q_{n-1}}$)
1	3	9	-
2	5	45	5
3	8	216	4.87
4	13	1053	4.84
5	21	5103	4.85
6	34	24786	4.85
7	55	120285	4.85
8	89	583929	4.85
9	144	2834352	4.85
10	233	13758417	4.85
⋮	⋮	⋮	⋮

Cuadro A.3: Cálculos para 3 capas con 2 opuestas

A.4. Cuatro capas, de las cuales 2 opuestas, $\langle R, L, F, U \rangle$

Vamos a realizar algo muy similar al del caso de $\langle R, L, U \rangle$. Una vez más, r_n son las secuencias de n capas que comienzan por R , y s_n las que no empiezan por R . Algunos ejemplos de cadenas son:

A.4. Cuatro capas, de las cuales 2 opuestas, $\langle R, L, F, U \rangle$

$s_1 :$	r_1
U	R
F	
L	
<hr/>	
$s_2 :$	r_2
UR	RU
FR	RF
LR	RL
LU	
FU	
UF	
LF	
UL	
FL	
\vdots	\vdots

La generación es parecida a la del anterior caso. Para generar nuevos casos de r_n y cumplir las reglas de las secuencias canónicas, se toman los elementos s_{n-1} y se añade una R al inicio. Para generar los de s_n , tomamos los de r_{n-1} y le añadimos una U o una R al inicio (generando $2r_{n-1}$ nuevos elementos). Después, tomamos los de s_{n-1} y de las 3 letras posibles, añadimos las 2 que no coincidan con el primer elemento. Así, generamos $2s_{n-1}$ elementos. Por tanto, las sucesiones tienen la siguiente forma:

$$\begin{cases} s_n = 2s_{n-1} + 2r_{n-1} \\ r_n = s_{n-1} \\ s_1 = 3 \\ r_1 = 1 \end{cases}$$

Operando, tenemos que:

$$s_n = 2(s_{n-1} + r_{n-1}) = 2(s_{n-1} + s_{n-2})$$

Finalmente, el número de secuencias canónicas es

$$q_n = 3^n(r_n + s_n) = 3^n(s_n + s_{n-1}) = \frac{3^n}{2}s_{n+1}$$

En este momento, ya podemos generar valores, y obtener un factor de ramificación de 8.19 (ver tabla A.4).

Cabe destacar que se ha programado un generador de secuencias canónicas en *Haskell*, gracias al cual se han corroborado experimentalmente todos estos números. Una versión simplificada del código se muestra a continuación:

Capítulo A. Cálculo de los factores de ramificación de algunos subgrupos

n giros	s_n	q_n secuencias canónicas	Factor de ramificación ($\frac{q_n}{q_{n-1}}$)
1	3	12	-
2	8	99	8,25
3	22	810	8,18
4	60	6642	8,2
5	164	54432	8,19
6	448	446148	8,19
7	1224	3656664	8,19
8	3344	29970648	8,19
9	9136	245643840	8,19
10	24960	2013334704	8,19
⋮	⋮	⋮	⋮

Cuadro A.4: Cálculos para 4 capas con 2 opuestas

```

numberCanonicalSequences :: Int -> [Face] -> [(Int, Int)]
numberCanonicalSequences n faces =
  map (\m -> (m, length (genAllCanonics m turns))) [1 .. n]

where
  turns = [Turn(f, m) | f <- faces, m <- [1..3]]

  genAllCanonics :: Int -> [Turn] -> [[Turn]]
  genAllCanonics 1 xs = map (\m -> [m]) xs
  genAllCanonics nMoves xs =
    concat (map (\m -> addOneMove m minusOne) xs)
  where
    minusOne = genAllCanonics (nMoves-1) xs

  addOneMove :: Turn -> [[Turn]] -> [[Turn]]
  addOneMove m xs = map (\moves -> (m : moves)) sublist
  where
    sublist = filter
      (\alg -> canJoinMoves m (head alg))

  canJoinMoves :: Turn -> Turn -> Bool
  canJoinMoves t1 t2 = (axisOfFace f1 /= axisOfFace f2)
    || f1 < f2
  where
    Turn (f1,_) = t1
    Turn (f2,_) = t2

```

Apéndice B

Código

A continuación se muestran los ficheros más importantes del código. Una versión actualizada puede encontrarse en mi GitHub:

https://github.com/Alexsql/Real_Bandaged_Cube_Explorer.

B.1. Programas de interacción con el usuario

```
module Main where

import InputAndSolve

main :: IO ()
main = do
    InputAndSolve.inputAndSolve

module InputAndSolve(inputAndSolve) where

import Data.Maybe(fromJust, fromMaybe)
import Bandaged
import Visualizer
import InputCube
import Moves(Algorithm(..))
import SolvingStrategies

inputAndSolve :: IO ()
inputAndSolve = do

    (bc, scheme) <- bandagedCubeScratchIO
    manimRecomendedVisualizer (stdCube bc) scheme (Algorithm [])

    let solution = smartKorfSolver bc
    let (Algorithm moves) = fromMaybe (Algorithm[]) solution
```

Capítulo B. Código

```
putStrLn ("\\n\\nSolution found: " ++ (show $ fromJust
solution) ++
"\n" ++ (show (length moves)) ++ " moves" ++
"\n\\n")

manimRecomendedVisualizer (stdCube bc) scheme (fromJust
solution)
```

B.2. Módulo Data

B.2.1. Fichero Cube.hs

```
module Cube (Cube(..), newCubeFromList, solved, slicePieces,
corners, edges, centers, allPieces) where

import Data.Group
import qualified Data.Vector.Unboxed as V
import Data.List(sortBy)

import Test.QuickCheck

newtype Cube = Cube (V.Vector Int) deriving (Show, Eq)

-- | Creates a cube by a 0-53 stickers list
newCubeFromList :: [Int] -> Cube
newCubeFromList xs = Cube $ V.fromList xs

instance Semigroup Cube where
  (Cube v1) <> (Cube v2) = Cube(V.unsafeBackpermute v1 v2)

instance Monoid Cube where
  mempty = Cube (V.fromList [0..53])

-- | Checks if a cube is solved
solved :: Cube -> Bool
solved = (== mempty)

instance Group Cube where
  invert (Cube xs) = Cube(V.fromList (invert_perm (V.toList
xs)))
```



```
invert_perm :: [Int] -> [Int]
invert_perm xs = map fst tups_ord
  where
    neutral = [0 .. ]
    tups = zip (neutral) xs
    tups_ord = sortBySnd tups
```

```

sortBySnd :: Ord b => [(a,b)] -> [(a,b)]
sortBySnd = sortBy (\(_, s1) (_, s2) -> compare s1 s2)

instance Ord Cube where
  compare (Cube v1) (Cube v2) = compare v1 v2

-- | Given a cube and a list, returns a list with all the
-- pieces of the position given.
slicePieces :: [Int] -> Cube -> [Int]
slicePieces ys (Cube xs) = V.toList (V.backpermute xs
  (V.fromList ys))

-- | Returns a list of the corner pieces
corners :: Cube -> [Int]
corners = slicePieces [0 .. 23]

-- | Returns a list of the edges pieces
edges :: Cube -> [Int]
edges = slicePieces [24 .. 47]

-- | Returns a list of the center pieces
centers :: Cube -> [Int]
centers = slicePieces [48 .. 53]

-- | Returns a list with all of the pieces
allPieces :: Cube -> [Int]
allPieces (Cube xs) = V.toList xs

instance Arbitrary Cube where
  arbitrary = do
    xs <- shuffle [0..53]
    return $ newCubeFromList xs

```

B.2.2. Fichero Moves.hs

```

module Moves (Turn(..), Algorithm(..), Face(..), Axis(..),
  algToPerm, possibleTurns, permOfTurn, lengthAlg, axisOfFace,
  isCanonicalSecuence) where

import Data.Group
import Cube

import Test.QuickCheck

-- | A face of a Cube (R, U, F, L, D, B)
data Face = N | R | U | F | L | D | B deriving(Show, Eq, Ord,
  Read, Enum)

```

Capítulo B. Código

```
-- / A Turn is a face-move in OBTM (tuple of Face and Int)
newtype Turn = Turn(Face, Int) deriving (Eq)

instance Show Turn where
    show (Turn(face, degrees))
        | face == N = ""
        | degrees == 2 = (show face) ++ "2"
        | degrees == 3 = (show face) ++ "'"
        | otherwise = (show face)

instance Read Turn where
    readsPrec _ (x:y:rest) =
        let move = read [x] :: Face
            num = read [y] :: Int
        in [ (simpOneTurn (Turn(move, num)) , rest) ]
    readsPrec _ _ = []

instance Read Algorithm where
    readsPrec _ str = [( (staticReadAlg (canonic str)), [] )]

staticReadAlg :: String -> Algorithm
staticReadAlg "" = Algorithm []
staticReadAlg (x:y:xs) = Algorithm [read ([x] ++ [y])] <>
    staticReadAlg xs
staticReadAlg _ = Algorithm []

canonic :: String -> String
canonic str = insertOnes strPrimes
    where
        strNoSpaces = filter (/= ' ') str
        strPrimes = map (\x -> if (x == '\'') then '3' else x)
        strNoSpaces

insertOnes :: String -> String
insertOnes "" = ""
insertOnes (x:y:xs) = if ((isCharMove x && isCharMove y)) then
    ([x] ++ "1" ++ insertOnes(y:xs)) else ([x] ++ insertOnes
    (y:xs))
insertOnes (x:xs) = if (isCharMove x) then ([x] ++ "1" ++
    insertOnes xs) else ([x] ++ insertOnes xs)

isCharMove :: Char -> Bool
isCharMove str = elem str (show [N .. ])

-- / Makes a list with all the possible turns
possibleTurns :: [Turn]
possibleTurns = [Turn(turn, degrees) | turn <- [R ..], degrees]
```

```

<- [1..3]

-- / An Algorithm is a list of moves
data Algorithm = Algorithm [Turn] deriving (Eq)

instance Show Algorithm where
  show (Algorithm xs) = concat (map (\x -> (show x) ++ " "))
    xs

instance Semigroup Algorithm where
  (Algorithm xs1) <> (Algorithm xs2) = Algorithm
    (simplifyTurns (xs1 ++ xs2))

simplifyTurns :: [Turn] -> [Turn]
simplifyTurns xs
| (length simplifiedVersion) < (length xs) = simplifyTurns
  simplifiedVersion
| otherwise = simplifiedVersion
where
  canonList = map simpOneTurn xs
  simplifiedVersion = simpAdjacentTurns canonList

simpAdjacentTurns :: [Turn] -> [Turn]
simpAdjacentTurns = foldl' myFunctionConcat []
where
  myFunctionConcat =
    (\xs a ->
      if (null xs)
        then [a]
      else
        ((init xs) ++ (simpTwoTurns (last xs) a))
    )

simpTwoTurns :: Turn -> Turn -> [Turn]
simpTwoTurns (Turn(face1, deg1)) (Turn(face2, deg2))
| face1 == face2 && cancel = []
| face1 == face2 = [simpOneTurn $ Turn(face1, sumD)]
| otherwise = [Turn(face1, deg1), Turn(face2, deg2)]
where
  sumD = deg1 + deg2
  cancel = sumD `mod` 4 == 0

simpOneTurn :: Turn -> Turn
simpOneTurn (Turn(f, degree))
| m == 0 = Turn(N, 0)
| otherwise = Turn(f, m)
where

```

Capítulo B. Código

```
m = degree `mod` 4

instance Monoid Algorithm where
  mempty = Algorithm []

instance Group Algorithm where
  invert (Algorithm (xs)) = Algorithm (invertListOfTurns xs)

  invertTurn :: Turn -> Turn
  invertTurn (Turn(f, n)) = Turn (f, (4-n))

  invertListOfTurns :: [Turn] -> [Turn]
  invertListOfTurns [] = []
  invertListOfTurns (x : xs) = invertListOfTurns xs ++
    [invertTurn x]

-- | Calculates the permutation an algorithm executes
algToPerm :: Algorithm -> Cube
algToPerm (Algorithm xs) = mconcat (map permOfTurn xs)

-- | Calculates the permutation a move executes
permOfTurn :: Turn -> Cube
permOfTurn (Turn(N, _)) = newCubeFromList [0..53]
permOfTurn (Turn(R, 1)) = newCubeFromList [0,1,2,3,4,5,11,9,10,
  13,14,12,17,15,16,7,8,6, 18,19,20,21,22,23,24,25,26,
  27,34,35,30,31,32,33,42,43, 28,29,38,39,40,41,36,37,44,
  45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(R, 2)) = newCubeFromList
  [0,1,2,3,4,5,12,13,14, 15,16,17,6,7,8,9,10,11,
  18,19,20,21,22,23,24,25,26, 27,42,43,30,31,32,33,36,37,
  34,35,38,39,40,41,28,29,44, 45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(R, 3)) = newCubeFromList
  [0,1,2,3,4,5,17,15,16, 7,8,6,11,9,10,13,14,12,
  18,19,20,21,22,23,24,25,26, 27,36,37,30,31,32,33,28,29,
  42,43,38,39,40,41,34,35,44, 45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(U, 1)) = newCubeFromList [9,10,11,0,1,2,3,4,5,
  6,7,8,12,13,14,15,16,17, 18,19,20,21,22,23,30,31,24,
  25,26,27,28,29,32,33,34,35, 36,37,38,39,40,41,42,43,44,
  45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(U, 2)) = newCubeFromList [6,7,8,9,10,11,0,1,2,
  3,4,5,12,13,14,15,16,17, 18,19,20,21,22,23,28,29,30,
  31,24,25,26,27,32,33,34,35, 36,37,38,39,40,41,42,43,44,
  45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(U, 3)) = newCubeFromList [3,4,5,6,7,8,9,10,11,
  0,1,2,12,13,14,15,16,17, 18,19,20,21,22,23,26,27,28,
  29,30,31,24,25,32,33,34,35, 36,37,38,39,40,41,42,43,44,
  45,46,47,48,49,50,51,52,53]
```

```

permOfTurn (Turn(F, 1)) = newCubeFromList
[22,23,21,3,4,5,6,7,8, 2,0,1,10,11,9,15,16,17,
18,19,20,14,12,13,24,25,26, 27,28,29,33,32,41,40,31,30,
36,37,38,39,35,34,42,43,44, 45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(F, 2)) = newCubeFromList
[12,13,14,3,4,5,6,7,8, 21,22,23,0,1,2,15,16,17,
18,19,20,9,10,11,24,25,26, 27,28,29,40,41,34,35,32,33,
36,37,38,39,30,31,42,43,44, 45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(F, 3)) = newCubeFromList [10,11,9,3,4,5,6,7,8,
14,12,13,22,23,21,15,16,17, 18,19,20,2,0,1,24,25,26,
27,28,29,35,34,31,30,41,40, 36,37,38,39,33,32,42,43,44,
45,46,47,48,49,50,51,52,53]
permOfTurn (Turn(L, 1)) = newCubeFromList
[5,3,4,19,20,18,6,7,8, 9,10,11,12,13,14,15,16,17,
23,21,22,1,2,0,38,39,26, 27,28,29,30,31,24,25,34,35,
36,37,46,47,40,41,42,43,44, 45,32,33,48,49,50,51,52,53]
permOfTurn (Turn(L, 2)) = newCubeFromList
[18,19,20,21,22,23,6,7,8, 9,10,11,12,13,14,15,16,17,
0,1,2,3,4,5,46,47,26, 27,28,29,30,31,38,39,34,35,
36,37,32,33,40,41,42,43,44, 45,24,25,48,49,50,51,52,53]
permOfTurn (Turn(L, 3)) = newCubeFromList
[23,21,22,1,2,0,6,7,8, 9,10,11,12,13,14,15,16,17,
5,3,4,19,20,18,32,33,26, 27,28,29,30,31,46,47,34,35,
36,37,24,25,40,41,42,43,44, 45,38,39,48,49,50,51,52,53]
permOfTurn (Turn(D, 1)) = newCubeFromList [0,1,2,3,4,5,6,7,8,
9,10,11,21,22,23,12,13,14, 15,16,17,18,19,20,24,25,26,
27,28,29,30,31,32,33,34,35, 36,37,38,39,46,47,40,41,42,
43,44,45,48,49,50,51,52,53]
permOfTurn (Turn(D, 2)) = newCubeFromList [0,1,2,3,4,5,6,7,8,
9,10,11,18,19,20,21,22,23, 12,13,14,15,16,17,24,25,26,
27,28,29,30,31,32,33,34,35, 36,37,38,39,44,45,46,47,40,
41,42,43,48,49,50,51,52,53]
permOfTurn (Turn(D, 3)) = newCubeFromList [0,1,2,3,4,5,6,7,8,
9,10,11,15,16,17,18,19,20, 21,22,23,12,13,14,24,25,26,
27,28,29,30,31,32,33,34,35, 36,37,38,39,42,43,44,45,46,
47,40,41,48,49,50,51,52,53]
permOfTurn (Turn(B, 1)) = newCubeFromList
[0,1,2,8,6,7,16,17,15, 9,10,11,12,13,14,20,18,19,
4,5,3,21,22,23,24,25,37, 36,28,29,30,31,32,33,34,35,
45,44,27,26,40,41,42,43,39, 38,46,47,48,49,50,51,52,53]
permOfTurn (Turn(B, 2)) = newCubeFromList
[0,1,2,15,16,17,18,19,20, 9,10,11,12,13,14,3,4,5,
6,7,8,21,22,23,24,25,44, 45,28,29,30,31,32,33,34,35,
38,39,36,37,40,41,42,43,26, 27,46,47,48,49,50,51,52,53]
permOfTurn (Turn(B, 3)) = newCubeFromList
[0,1,2,20,18,19,4,5,3, 9,10,11,12,13,14,8,6,7,
16,17,15,21,22,23,24,25,39, 38,28,29,30,31,32,33,34,35,

```

Capítulo B. Código

```
27, 26, 45, 44, 40, 41, 42, 43, 37, 36, 46, 47, 48, 49, 50, 51, 52, 53]
permOfTurn (Turn(t, x)) = permOfTurn(Turn(t, x `mod` 4))

-- | Returns the length of an algorithm
lengthAlg :: Algorithm -> Int
lengthAlg (Algorithm xs) = length xs

-- | Returns if a sequence is canonical
isCanonicalSecuence :: [Turn] -> Bool
isCanonicalSecuence xs = canAux (Turn(N, 0) : xs)
  where
    canAux :: [Turn] -> Bool
    canAux [] = True
    canAux [_] = True
    canAux (Turn(x, _) : Turn(y, m) : rest) = ((axisOfFace x /=
      axisOfFace y) || x < y)
      && canAux (Turn(y, m) : rest)

-- | Axis of the cube
data Axis = RL | UD | FB | NN deriving (Show, Eq, Ord, Read,
  Enum)

-- | Returns the axis of a Face
axisOfFace :: Face -> Axis
axisOfFace R = RL
axisOfFace L = RL
axisOfFace U = UD
axisOfFace D = UD
axisOfFace F = FB
axisOfFace B = FB
axisOfFace N = NN

instance Arbitrary Turn where
  arbitrary = elements possibleTurns

instance Arbitrary Algorithm where
  arbitrary = do
    xs <- listOf arbitrary
    return (Algorithm xs <> mempty)
```

B.2.3. Fichero Bandaged.hs

```
module Bandaged(BandagedCube(..), solvedBC, deleteBlocks,
  tryToTurn, tryToExecuteAlg, validTurn, unsafeExecutionAlg,
  divideTurn) where

import Cube
import Moves
```

```

import qualified Data.Set as S
import Data.Maybe(isNothing, fromJust)
import Data.List(intercalate)

import Test.QuickCheck

-- | A Bandaged Cube is a Cube with a set of sets of
-- restrictions.

data BandagedCube = BandagedCube {stdCube :: Cube, restrictions
  :: S.Set (S.Set Int)} deriving Eq

instance Ord BandagedCube where
  compare (BandagedCube c1 _) (BandagedCube c2 _) = compare
    c1 c2

instance Show BandagedCube where
  show (BandagedCube cube setRestrictions) = (show cube) ++
    "\n" ++
    (show listRestrictions) ++
    "\n" ++
    --(show listRestrictions) ++
    "\n" ++
    "BLOCKS: \n" ++
    (intercalate "\n" (map unwords
      facesPieces))

  where
    listRestrictions = S.toList (S.map S.toList
      setRestrictions)
    notRepeatingPieces = map (filterEachBlock)
      listRestrictions
    facesPieces = (map . map) intToStrPiece
      notRepeatingPieces

    filterEachBlock :: [Int] -> [Int]
    filterEachBlock = filter (\x -> (x < 24 && (x `mod` 3 == 0)) ||
      (x >= 24 && x < 48 && (x `mod` 2 == 0)) ||
      (x >= 48))

intToStrPiece :: Int -> String
intToStrPiece n
  | n < 24 = cornersP !! (n `div` 3)
  | n < 48 = edgesP !! ((n - 24) `div` 2)
  | otherwise = centersP !! (n - 48)
  where
    cornersP = ["UFL", "UBL", "UBR", "UFR", "DFR", "DBR",
      "DBL", "DFL"]
    edgesP = ["UL", "UB", "UR", "UF",
      "FL", "FR", "BR", "BL",
      "DF", "DR", "DB", "DL"]
    centersP = ["U", "F", "R", "B", "L", "D"]

```

Capítulo B. Código

```
solvedBC :: BandagedCube -> Bool
solvedBC (BandagedCube c _) = solved c

-- | Given a Bandaged Cube, returns the cube without blocks
deleteBlocks :: BandagedCube -> Cube
deleteBlocks bc = stdCube bc

-- | Execute an algorithm on a Bandaged Cube when it does not
-- break any block
tryToExecuteAlg :: BandagedCube -> Algorithm -> Maybe
BandagedCube
tryToExecuteAlg bc (Algorithm xsmoves) = moveByMove (Just bc)
xsmoves
where
  moveByMove :: Maybe BandagedCube -> [Turn] -> Maybe
  BandagedCube
  moveByMove Nothing _ = Nothing
  moveByMove bcAux [] = bcAux
  moveByMove (Just bcAux) (x:xs) = let next = tryToTurn
  bcAux x
  in moveByMove next xs

-- | Try to execute an algorithm on a Bandaged Cube. When a
-- move is not valid, it skips the move. It returns the final
-- state and the algorithm executed.
unsafeExecutionAlg :: BandagedCube -> Algorithm ->
(BandagedCube, Algorithm)
unsafeExecutionAlg bc (Algorithm moves) = (finalCube, Algorithm
(reverse xsMoves))
where
  (finalCube, xsMoves) = moveByMove bc moves []
  moveByMove :: BandagedCube -> [Turn] -> [Turn] ->
(BandagedCube, [Turn])
  moveByMove bcAux [] accMoves = (bcAux, accMoves)
  moveByMove bcAux (x:xs) accMoves
  | isNothing next = moveByMove bcAux xs accMoves
  | otherwise = moveByMove (fromJust next) xs (x :
accMoves)
  where
    next = tryToTurn bcAux x

-- | Makes a move when it is possible and does not break any
-- block
tryToTurn :: BandagedCube -> Turn -> Maybe BandagedCube
tryToTurn bCube currTurn
```

```

| validTurn bCube f = Just (BandagedCube {stdCube =
  newPerm, restrictions = restr})
| otherwise = Nothing
where
  (BandagedCube currCube restr) = bCube
  (Turn(f, _)) = currTurn
  newPerm = currCube <> (permOfTurn currTurn)

-- | Checks if a Face would not break any block
validTurn :: BandagedCube -> Face -> Bool
validTurn bCube face = all (checkOneBlock) allRestr
where
  allRestr = restrictions bCube
  checkOneBlock = turnPreserveBlock bCube face

turnPreserveBlock :: BandagedCube -> Face -> S.Set Int -> Bool
turnPreserveBlock (BandagedCube cubeState _) face block =
  (S.disjoint block s1Real) || (S.disjoint block s2Real)
where
  (s1, s2) = divideTurn face
  s1Real = S.fromList (slicePieces (S.toList s1)
    cubeState)
  s2Real = S.fromList (slicePieces (S.toList s2)
    cubeState)

-- | Given a face, returns a tuple with the pieces afected and
-- not afected respectively.
divideTurn :: Face -> (S.Set Int, S.Set Int)
divideTurn m = (piecesAfected m, piecesNotAfected m)

piecesAfected :: Face -> S.Set Int
piecesAfected R = S.fromList
  [6,7,8,9,10,11,12,13,14,15,16,17,28,29,34,35,36,37,42,43,50]
piecesAfected U = S.fromList
  [9,10,11,0,1,2,3,4,5,6,7,8,30,31,24,25,26,27,28,29,48]
piecesAfected F = S.fromList
  [22,23,21,2,0,1,10,11,9,14,12,13,33,32,41,40,31,30,35,34,49]
piecesAfected L = S.fromList
  [5,3,4,19,20,18,23,21,22,1,2,0,38,39,24,25,46,47,32,33,52]
piecesAfected D = S.fromList
  [21,22,23,12,13,14,15,16,17,18,19,20,46,47,40,41,42,43,44,45,53]
piecesAfected B = S.fromList
  [8,6,7,16,17,15,20,18,19,4,5,3,37,36,45,44,27,26,39,38,51]
piecesAfected _ = S.fromList []

piecesNotAfected :: Face -> S.Set Int
piecesNotAfected bm = S.difference s0_53 (piecesAfected bm)

```

Capítulo B. Código

```
where
s0_53 = S.fromList [0..53]

instance Arbitrary BandagedCube where
    arbitrary = do
        alg <- arbitrary
        let bcs = BandagedCube {stdCube = newCubeFromList
            [0..53],
            restrictions = S.singleton (S.empty)}
        return (fromJust (tryToExecuteAlg bcs alg))
```

B.2.4. Fichero MathematicalNotation.hs

```
module MathematicalNotation(cornerState, edgesState,
    edgesSplittedState) where

import Cube
import Bandaged(BandagedCube(..))

-- | Returns the state of corners in "mathematical" notation
cornerState :: BandagedCube -> ([Int], [Int])
cornerState (BandagedCube cube _) = (perm, ori)
    where
        xs = zip [0 .. 23] (corners cube)
        xs2 = filter (\t -> (snd t) `mod` 3 == 0) xs
        perm = map (\(_ ,x) -> x `div` 3) xs2
        ori = map (\(x,_) -> x `mod` 3) xs2

-- | Returns the state of edges in "mathematical" notation
edgesState :: BandagedCube -> ([Int], [Int])
edgesState (BandagedCube cube _) = (perm, ori)
    where
        xs = zip [24 .. 47] (edges cube)
        xs2 = filter (\(_ , y) -> (y `mod` 2 == 0)) xs
        perm = map (\(_ ,x) -> (x - 24) `div` 2) xs2
        ori = map (\(x,_) -> x `mod` 2) xs2

edgesSplittedState :: BandagedCube -> (([Int], [Int]),
    ([Int], [Int]))
edgesSplittedState bc = (s1, s2)
    where
        (p, o) = edgesState bc
        s1 = (take 6 p, take 6 o)
        s2 = (drop 6 p, drop 6 o)
```

B.3. Módulo de IO-Visualizator

B.3.1. Fichero ExpandBlocks.hs

```

module ExpandBlocks (expandBlocks) where

import Moves
import Bandaged
import qualified Data.Set as S

-- | Expand all the blocks
expandBlocks :: S.Set (S.Set Int) -> S.Set (S.Set Int)
expandBlocks setOfBlocks = S.map expand1Block setOfBlocks

--Expand a single block, with all the possible turns
expand1Block :: S.Set Int -> S.Set Int
expand1Block block = pieceIntersections (expansions)
where
    expansions = map (expandBlockTurn block) [R, U, L, D, B, F]

--Given a block and a Face, returns the expanded block
expandBlockTurn :: S.Set Int -> Face -> S.Set Int
expandBlockTurn block move
    | isBlockPreserved = rightSubSet
    | otherwise = s0_53
where
    (xs1, xs2) = divideTurn move
    --Every turn divides the cube in 2 sets: pieces
    --affected and not affected.
    --The block is preserved when the full block is in one
    --of the 2 sub-partitions
    isBlockPreserved = (S.disjoint block xs1) ||
        (S.disjoint block xs2)

    --There are going to be intersections. Must return the
    --proper sub-partition
    rightSubSet = if (S.disjoint block xs2) then xs1 else
        xs2

pieceIntersections :: (Foldable f) => f (S.Set Int) -> S.Set Int
pieceIntersections = foldl' S.intersection (full)
where
    full = s0_53

-- | A set of numbers 0..53
s0_53 :: S.Set Int
s0_53 = S.fromList [0..53]

```

Capítulo B. Código

B.3.2. Fichero InputBandagedCube.hs

```
module InputBandagedCube(newBandagedCube,
    newSolvedBandagedCube, newSolvedCube) where

import Cube
import Bandaged
import ExpandBlocks(expandBlocks)
import qualified Data.Set as S

-- | Creates a standard cube
newSolvedCube :: Cube
newSolvedCube = newCubeFromList [0..53]

-- | Creates a solved Bandaged with no blocks
newSolvedBandagedCube :: BandagedCube
newSolvedBandagedCube = newBandagedCube (newCubeFromList
    [0..53]) []

-- | Creates a new Bandaged Cube, with a standard cube and the
-- restrictions
newBandagedCube :: Cube -> [[Int]] -> BandagedCube
newBandagedCube cubeOrigin blocks = BandagedCube {stdCube =
    cubeOrigin, restrictions = postProcessedBlocks}
    where
        blocksSet = doubleListToDoubleSet blocks
        postProcessedBlocks = (expandBlocks . canonicSets)
            blocksSet

--All auxiliary from here

doubleListToDoubleSet :: Ord a => [[a]] -> S.Set (S.Set a)
doubleListToDoubleSet xs = S.fromList (map S.fromList xs)

-- | Join (union) the sets that are not disjoint.
canonicSets :: S.Set (S.Set Int) -> S.Set (S.Set Int)
canonicSets ss = genUnions (S.toList ss) (doubleListToDoubleSet
    [[]])

----- Auxiliar of canonic set. Takes a list of sets, makes the
----- unions recursively
genUnions :: [S.Set Int] -> S.Set (S.Set Int) -> S.Set (S.Set
    Int)
genUnions [] xs = S.delete (S.empty) xs
genUnions (s0:rest) temp = genUnions rest (S.insert unitedSet
    notCombining)
    where
        (notCombining, toComb) = S.partition (S.disjoint s0)
```

```
    temp
unitedSet = S.unions (S.insert s0 toComb)
```

B.3.3. Fichero ManimHsConversion.hs

```
module ManimHsConversion(cubeFromManimCodification,
  toManimCodification, facePieceToInts, swapByEquivalent) where

import Cube
import Moves(Face(..))

-- | Given a cube, returns the cube in Manim codification
-- (initial of face in its order)
toManimCodification :: Cube -> String
toManimCodification cube = foldl' (\acc face -> acc ++ (show
  face)) "" listFaces
  where
    permutation = [3,26,6,24,48,28,0,30,9,
      10,29,8,35,50,37,14,43,16,
      1,31,11,32,49,34,23,41,13,21,40,12,46,53,42,18,44,15,
      4,25,2,39,52,33,20,47,22,7,27,5,36,51,38,17,45,19]
    xs = allPieces cube
    reorder = map (xs !!) permutation
    listFaces = map pieceToFace reorder

pieceToFace :: Int -> Face
pieceToFace n = xs !! n
  where
    xs = [U,F,L,U,L,B,U,B,R, U,R,F,D,F,R,D,R,B,
      D,B,L,D,L,F,U,L,U, B,U,R,U,F,F,L,F,R,
      B,R,B,L,D,F,D,R,D, B,D,L,U,F,R,B,L,D]

-- | Returns the cube given by an equivalence of colours and a
-- list in manim's order
cubeFromManimCodification :: [(String, String)] -- ^
  Equivalence, like [(\"U\", \"White\"), (\"F\", \"Green\"),
  (\"R\", \"Red\"), (\"L\", \"Orange\"), (\"B\", \"Blue\"),
  (\"D\", \"Yellow\")]
  -> [String] -- ^ List of
    colours, like [\"White\", \"Green\", ...]
  -> Cube

cubeFromManimCodification equivalence str = newCubeFromList perm
  where
    perm = (stringToNum . reorder . swapByEquivalent
      equivalence) str
    reorder :: [String] -> [String]
    reorder xs = map (xs !!) arrangement
```

Capítulo B. Código

```
where
  arrangement = [6,18,38,0,36,47,2,45,11,
  8,9,20,29,26,15,35,17,51,
  33,53,42,27,44,24,3,37,1,
  46,5,10,7,19,21,41,23,
  12,48,14,50,39,28,25,32,16,
  34,52,30,43,4,22,13,49,40,31]

-- | Given tuples of equivalences, makes all the equivalences.
swapByEquivalent :: [(String, String)] -> [String] -> [String]
swapByEquivalent eq xs = map (takeEquiv eq) xs

-- | Given tuples of equivalences, returns the equivalent entry
takeEquiv :: Eq a => [(a,a)] -> a -> a
takeEquiv [] n = n
takeEquiv ((x,y):xs) current
| y == current = x
| x == current = y
| otherwise = takeEquiv xs current

-- | given ["URF", "UR", "UL"] returns the ints values [0, 1,
2, 24, 25, 28, 29]
facePieceToInts :: [String] -> [Int]
facePieceToInts str
| length str == 1 = centerToInt str
| length str == 2 = edgeToInts str
| length str == 3 = cornerToInts str
| otherwise = []

stringToNum :: [String] -> [Int]
stringToNum xs = (mapBy 3 cornerToInts cornersP) ++ (mapBy 2
  edgeToInts edgesP) ++ (mapBy 1 centerToInt centersP)
  where
    cornersP = take 24 xs
    edgesP = (take 24 . drop 24) xs
    centersP = drop 48 xs

mapBy :: Int -> ([a] -> [b]) -> [a] -> [b]
mapBy _ _ [] = []
mapBy n f xs = (f prefix) ++ (mapBy n f (drop n xs))
  where
    prefix = take n xs

centerToInt :: [String] -> [Int]
centerToInt ["U"] = [48]
centerToInt ["F"] = [49]
centerToInt ["R"] = [50]
```

```

centerToInt ["B"] = [51]
centerToInt ["L"] = [52]
centerToInt ["D"] = [53]
centerToInt _ = [-1]

edgeToInts :: [String] -> [Int]
edgeToInts [] = []
edgeToInts [_] = []
edgeToInts (f1 : f2 : _) 
  | (not . null) opt1 = opt1
  | (not . null) opt2 = opt2
  | otherwise = []

where
  opt1 = tryToMatch f1 f2
  opt2 = reverse (tryToMatch f2 f1)

--tryToMatch always returns ascending lists
tryToMatch :: String -> String -> [Int]
tryToMatch "U" "L" = [24, 25]
tryToMatch "U" "B" = [26, 27]
tryToMatch "U" "R" = [28, 29]
tryToMatch "U" "F" = [30, 31]

tryToMatch "F" "L" = [32, 33]
tryToMatch "F" "R" = [34, 35]
tryToMatch "B" "R" = [36, 37]
tryToMatch "B" "L" = [38, 39]

tryToMatch "D" "F" = [40, 41]
tryToMatch "D" "R" = [42, 43]
tryToMatch "D" "B" = [44, 45]
tryToMatch "D" "L" = [46, 47]

tryToMatch _ _ = []

cornerToInts :: [String] -> [Int]
cornerToInts [] = []
cornerToInts [_] = []
cornerToInts [_,_] = []
cornerToInts (f1 : f2 : f3 : _) 
  | (not . null) opt1 = opt1
  | (not . null) opt2 = (opt2 !! 2) : (opt2 !! 0) : (opt2 !!
  1) : []
  | (not . null) opt3 = (opt3 !! 1) : (opt3 !! 2) : (opt3 !!
  0) : []
  | (not . null) opt4 = (opt4 !! 1) : (opt4 !! 0) : (opt4 !!
  1)

```

Capítulo B. Código

```
2) : []
| (not . null) opt5 = (opt5 !! 0) : (opt5 !! 2) : (opt5 !!
1) : []
| (not . null) opt6 = (opt6 !! 2) : (opt6 !! 1) : (opt6 !!
0) : []
| otherwise = []
where
  opt1 = tryToMatch f1 f2 f3
  opt2 = tryToMatch f2 f3 f1
  opt3 = tryToMatch f3 f1 f2

  opt4 = tryToMatch f1 f3 f2
  opt5 = tryToMatch f3 f2 f1
  opt6 = tryToMatch f2 f1 f3
  --tryToMatch always returns ascending lists
  tryToMatch :: String -> String -> String -> [Int]
  tryToMatch "U" "F" "L" = [0, 1, 2]
  tryToMatch "U" "L" "B" = [3, 4, 5]
  tryToMatch "U" "B" "R" = [6, 7, 8]
  tryToMatch "U" "R" "F" = [9, 10, 11]

  tryToMatch "D" "F" "R" = [12, 13, 14]
  tryToMatch "D" "R" "B" = [15, 16, 17]
  tryToMatch "D" "B" "L" = [18, 19, 20]
  tryToMatch "D" "L" "F" = [21, 22, 23]
  tryToMatch _ _ _ = []
```

B.3.4. Fichero InputCube.hs

```
module InputCube(bandagedCubeScratchIO) where

import Cube
import Bandaged
import InputBandagedCube
import ManimHsConversion

import Data.List.Split(splitOn)
import Data.List(intercalate)

-- | Definitive IO for asking the user to generate a Bandaged
-- Cube
bandagedCubeScratchIO :: IO (BandagedCube, String)
bandagedCubeScratchIO = do

  --Input of colours
  equivils <- faceAliases
  sch <- colourScheme equivils
```

```

--Input of the cube
cube <- input54Stickers equivs

--Input of the blocks
xs <- inputBlock equivs

return $ (newBandagedCube cube xs, sch)

--Example of list of tuples: [("U", "White"), ("F", "Green"),
("R", "Red"), ("L", "Orange"), ("B", "Blue"), ("D",
"Yellow")]
-- / An IO that guides the user to insert the face aliases
faceAliases :: IO[(String, String)]
faceAliases = do
  putStrLn "Insert alias for U face (default: w)"
  uLayer <- getLine
  putStrLn "Insert alias for R face (default: r)"
  rLayer <- getLine
  putStrLn "Insert alias for F face (default: g)"
  fLayer <- getLine
  putStrLn "Insert alias for D face (default: y)"
  dLayer <- getLine
  putStrLn "Insert alias for L face (default: g)"
  lLayer <- getLine
  putStrLn "Insert alias for B face (default: b)"
  bLayer <- getLine

  let defaultInput = [("U", "w"), ("R", "r"), ("F", "g"),
  ("L", "o"), ("B", "b"), ("D", "y")]
  let input = [("U", uLayer), ("R", rLayer), ("F", fLayer),
  ("L", lLayer), ("B", bLayer), ("D", dLayer)]
  let sol = zipWith (\(i, alias) (i2, def) -> if(length alias
  == 0) then (i2, def) else (i, alias)) input defaultInput

return (sol)

-- / An IO that guides the user to insert a cube
input54Stickers :: [(String, String)] -> IO Cube
input54Stickers equivs = do
  uLayer <- oneFace "U" equivs
  rLayer <- oneFace "R" equivs
  fLayer <- oneFace "F" equivs
  dLayer <- oneFace "D" equivs
  lLayer <- oneFace "L" equivs
  bLayer <- oneFace "B" equivs
return $ cubeFromManimCodification equivs (uLayer ++ rLayer

```

Capítulo B. Código

```
++ fLayer ++ dLayer ++ lLayer ++ bLayer)

-- | IO that asks for the stickers of 1 face (private)
oneFace :: String -> [(String, String)] -> IO [String]
oneFace face eq = do
  putStrLn ("Insert " ++ face ++ " (alias " ++ alias ++
  ") face colours, separated by spaces (default: " ++
  "defaultFace ++ ")" )
  stickers <- getLine

let provSol = ((filter (/= "")) . (splitOn " " ))
  (checkEmpty stickers)

if ((canBeValid provSol) && (provSol !! 4) == alias)
  then return (provSol)
  else (
    do
      putStrLn "Invalid input"
      strNext <- oneFace face eq
      return (strNext)
  )

where
  result = swapByEquivalent eq [face]
  alias = case result of
    [x] -> x
    _ -> error "Unexpected result length"
  defaultFace = (concat . replicate 9) (alias ++ " ")

  checkEmpty :: String -> String
  checkEmpty "" = defaultFace
  checkEmpty str = str

  canBeValid :: [String] -> Bool
  canBeValid str = (length str == 9) && (all (\x -> x
  `elem` (map snd eq)) str)

--"WHITE, #B90000, #009B48, #FFD500, #FF5900, #0045AD"
--"#FFFFFF, RED, GREEN, YELLOW, ORANGE, BLUE"
colourScheme :: [(String, String)] -> IO String
colourScheme equiv = do
  putStrLn "\nDefault hexadecimal colours: "
  putStrLn "White: #FFFFFF"
  putStrLn "Red: #B90000"
  putStrLn "Green: #009B48"
  putStrLn "Yellow: #FFD500"
  putStrLn "Orange: #FF5900"
```

```

putStrLn "Blue: #0045AD"
putStrLn ""

putStrLn ("Insert colour for U face (alias ""++ (xsAlias !!
  0) ++ "") (hexadecimal or by name, default #FFFFFF
  (white))")
0x <- getLine
putStrLn ("Insert colour for R face (alias ""++ (xsAlias !!
  1) ++ "") (hexadecimal or by name, default #B90000
  (red))")
0x <- getLine
putStrLn ("Insert colour for F face (alias ""++ (xsAlias !!
  2) ++ "") (hexadecimal or by name, default #009B48
  (green))")
0x <- getLine
putStrLn ("Insert colour for D face (alias ""++ (xsAlias !!
  3) ++ "") (hexadecimal or by name, default #FFD500
  (yellow))")
0x <- getLine
putStrLn ("Insert colour for L face (alias ""++ (xsAlias !!
  4) ++ "") (hexadecimal or by name, default #FF5900
  (orange))")
0x <- getLine
putStrLn ("Insert colour for B face (alias ""++ (xsAlias !!
  5) ++ "") (hexadecimal or by name, default #0045AD
  (blue))")
0x <- getLine
let xs = map checkEmpty [('U', 0x), ('R', 0x), ('F',
0x), ('D', 0x), ('L', 0x), ('B', 0x)]
return $ intercalate "," xs

where

  xsAlias = swapByEquivalent equiv ["U", "R", "F", "D",
  "L", "B"]

  checkEmpty :: (Char, String) -> String
  checkEmpty (face, "") = defaultColour face
  checkEmpty (_, ('#':colour)) = ('#':colour)
  checkEmpty (_, colour) = ('#':colour)

  defaultColour :: Char -> String
  defaultColour 'U' = "#FFFFFF"
  defaultColour 'R' = "#B90000"
  defaultColour 'F' = "#009B48"
  defaultColour 'D' = "#FFD500"
  defaultColour 'L' = "#FF5900"
  defaultColour 'B' = "#0045AD"

```

Capítulo B. Código

```
defaultColour _ = "#555555" --dark grey

-- / An IO that guides the user to insert a block
inputBlock :: [(String, String)] -> IO [[Int]]
inputBlock equiv = do
  putStrLn "(Optional) insert a block in the format:
  c1-c2-c3+c1-c2+c1 (empty for finish)"
  str <- getLine

  --Keep on asking for input until null
  if (null str)
    then (return [])
    else do
      rest <- inputBlock equiv
      return ([convertBlockToInts str] ++ rest)
  where
    convertBlockToInts :: String -> [Int]
    convertBlockToInts str
      | null str = []
      | otherwise = concat $ map facePieceToInts colours
      where
        pieces = (filter (/= ""))
        filter (/= ' ') str
        stickers = map (filter (/= ""))
        splitOn "-" pieces
        colours = map (swapByEquivalent equiv) stickers
```

B.3.5. Fichero Visualizator.hs

```
module Visualizator(manimCustomVisualizer,
                     manimRecomendedVisualizer) where

import Cube
import Moves
import ManimHsConversion(toManimCodification)
import System.Process (spawnProcess)

-- / Generates a video with manim, with recomended options for
-- rendering
manimRecomendedVisualizer :: Cube -- ^ Initial position
                           -- ^ Color scheme ("" for
                           default)
                           -- ^ Algorithm executed
                           -- IO()

manimRecomendedVisualizer cube "" algorithm = do
  manimCustomVisualizer 10 1.5 "low_quality" cube
  "WHITE,#B90000,#009B48,#FFD500,#FF5900,#0045AD" algorithm
```

```

manimRecomendedVisualizer cube scheme algorithm = do
    manimCustomVisualizer 10 1.5 "low_quality" cube scheme
    algorithm

-- / Generates a video with manim, allowing configuration
manimCustomVisualizer :: Int -- ^ Seconds rotating over the
diagonal
    -> Float -- ^ Seconds for each move
    -> String -- ^ "low_quality" or "high_quality"
    -> Cube -- ^ Initial position
    -> String -- ^ String of colours in order
        URFLDB. Example
        "WHITE, #B90000, #009B48, #FFD500, #FF5900, #0045AD"
    -> Algorithm -- ^ Algorithm executed
    -> IO()

manimCustomVisualizer tRotation tMoves quality cube scheme alg
= do
    _ <- spawnProcess "python"
        ["src-exe/IO_Visualizer/manim_cube_visualizer.py",
        show tRotation,
        show tMoves,
        quality,
        toManimCodification cube,
        scheme,
        show alg]
    putStrLn "Generating video with Manim"

```

B.4. Módulo Heuristics

B.4.1. Fichero Combinatorics.hs

```

module Combinatorics(factorialNumbering, nprNumbering) where

import Data.List(elemIndex, sort, delete)
import Data.Maybe(fromJust)

-- / Recieves a permutation of elements and returns its
numeration
factorialNumbering :: [Int] -> Int
factorialNumbering xp = factorialNumberingGlobal xp (sort xp)
where
    factorialNumberingGlobal :: [Int] -> [Int] -> Int
    factorialNumberingGlobal [] _ = 0
    factorialNumberingGlobal (x:xs) orig = thisElem +
        factorialNumberingGlobal xs xs2
    where
        (i, xs2) = firstOccurrence x orig

```

Capítulo B. Código

```
    thisElem = i * factorial (length xs)

factorial :: Int -> Int
factorial n = product [2 .. n]

-- / Recieves a variation of elements and returns its numeration
nprNumbering :: [Int]          -- ^ Total elements
              -> [Int]          -- ^ The variation
              -> Int

nprNumbering totalNumbers perm = npr perm totalNumbers 0 n (n -
r)
  where
    n = length totalNumbers
    r = length perm

npr :: [Int] -> [Int] -> Int -> Int -> Int -> Int
npr [] _ _ _ _ = 0
npr (x:xs) totalElems index cardTotalElems denom =
  thisElem + npr xs xs2 (index + 1) cardTotalElems
  denom
  where
    (i, xs2) = firstOccurrence x totalElems
    thisElem = i * (product [denom + 1 ..
    cardTotalElems - 1 - index])

firstOccurrence :: Int -> [Int] -> (Int, [Int])
firstOccurrence n xs = (i, xs2)
  where
    i = fromJust (elemIndex n xs)
    xs2 = delete n xs
```

B.4.2. Fichero IndexHeuristics.hs

```
module IndexHeuristics(cornersKey, edgesKey, edgesKeyFst,
edgesKeySnd) where

import Bandaged
import Combinatorics(nprNumbering, factorialNumbering)
import MathematicalNotation(cornerState, edgesState)
import Data.List(sortBy)

-- / Returns the key of the corners of a BCube
cornersKey :: BandagedCube -> Int
cornersKey bc = (permKey * 3 ^ (7 :: Int)) + orKey
  where
    (perm, ori) = cornerState bc
    permKey = factorialNumbering perm
    orKey = sum [(3 ^ i * ori !! (6 - i)) | i <- [0 .. 6]]
```

```

-- | Returns the key of the first 6 edges
edgesKeyFst :: BandagedCube -> Int
edgesKeyFst c = fst (edgesKey c)

-- | Returns the key of the second 6 edges
edgesKeySnd :: BandagedCube -> Int
edgesKeySnd c = snd (edgesKey c)

-- | Returns the key of the halves of the edges of BCube
edgesKey :: BandagedCube -> (Int, Int)
edgesKey bc = (keyFst, keySnd)
  where
    (perm, ori) = edgesState bc
    allp = zip3 [0..11] perm ori

    sortedEdges = sortBy (\( _, p1, _) ( _, p2, _) -> compare
      p1 p2) allp
    (iReorder, _, oReorder) = unzip3 sortedEdges

    keyFst = indexHalfE (take 6 iReorder) (take 6 oReorder)
    keySnd = indexHalfE (drop 6 iReorder) (drop 6 oReorder)

indexHalfE :: [Int] -> [Int] -> Int
indexHalfE perm0 ori0 = (permKey * 2 ^ (6 :: Int)) + orKey
  where
    permKey = nprNumbering [0..11] perm0
    orKey = sum [(2 ^ i * ori0 !! (5 - i)) | i <- [0 .. 5]]

```

B.4.3. Fichero GenKorfHeuristics.hs

```

module GenKorfHeuristics(lookupAll, stdVectors) where

import qualified Data.Set as S
import Data.Maybe(isJust, fromJust)
import Data.Word(Word8)

import Bandaged
import Moves
import InputBandagedCube(newSolvedBandagedCube)
import IndexHeuristics

import Data.PSQueue as PS
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as MV
import Control.Monad.ST
import Control.Monad(forM_)

```

Capítulo B. Código

```
-- / Alias for Word8 Vectors
type Vector8 = V.Vector Word8

-- / Calculates a vector with the depths of a pattern database
stdVectors :: (Vector8, Vector8, Vector8)
stdVectors = (c, e1, e2)
where
  maxDepth = 5
  c = cornersVector ini maxDepth
  e1 = edgesFstVector ini maxDepth
  e2 = edgesSndVector ini maxDepth
  ini = newSolvedBandagedCube

-- / Accesses the pattern database and return the minimum
  number of moves for each piece set
lookupAll :: BandagedCube -> (Word8, Word8, Word8)
lookupAll bc = (lookupCorners bc, lookupFstEdges bc,
  lookupSndEdges bc)

-- / Generate a pattern database of corners from a state to
  depth n
cornersVector :: BandagedCube           -- ^ Initial state
  (solved recommended)
    -> Word8                         -- ^ Maximum depth
    -> Vector8
cornersVector bc n = applyChangesMV 88179840 (n+1) ch
where
  ch = bfsStoreChanges cornersKey n [R .. ] bc

-- / Generate a pattern database of the first 6 edges from a
  state to depth n
edgesFstVector :: BandagedCube           -- ^ Initial state
  (solved recommended)
    -> Word8                         -- ^ Maximum depth
    -> Vector8
edgesFstVector bc n = applyChangesMV 42577920 (n+1) ch
where
  ch = bfsStoreChanges edgesKeyFst n [R .. ] bc

-- / Generate a pattern database of the last 6 edges from a
  state to depth n
edgesSndVector :: BandagedCube           -- ^ Initial state
  (solved recommended)
    -> Word8                         -- ^ Maximum depth
    -> Vector8
edgesSndVector bc n = applyChangesMV 42577920 (n+1) ch
where
```

```

ch = bfsStoreChanges edgesKeySnd n [R .. ] bc

-- / Make the immutable vector (with mutable operations)
applyChangesMV :: Int          -- ^ Size
              -> Word8          -- ^ Maximum depth
              -> [(Int, Word8)]    -- ^ Changes
              -> Vector8

applyChangesMV sizeV defaultDepth changes = runST $ do
  mv <- MV.replicate sizeV defaultDepth
  myUpdate mv changes
  V.unsafeFreeze mv

myUpdate :: MV.MVector#(Word8) -> [(Int, Word8)] -> ST#(s)()
myUpdate v changes = formM_ changes $ (\(i, value) -> MV.write v
  i value)

newtype GenerationState = GenerationState (Int, Face,
                                             BandagedCube)

instance Eq GenerationState where
  (GenerationState (key1, _, _)) == (GenerationState (key2,
  _, _)) = key1 == key2

instance Ord GenerationState where
  compare (GenerationState (key1, _, _)) (GenerationState (key2,
  _, _)) = compare key1 key2

instance Show GenerationState where
  show (GenerationState (k, _, _)) = '#' : (show k)

type SetVisitedKeys = S.Set#(Int)

bfsStoreChanges :: (BandagedCube -> Int) -> Word8 -> [Face] ->
  BandagedCube -> [(Int, Word8)]
bfsStoreChanges kGen maxDepth faces initBC = bfs kGen maxDepth
  (PS.singleton gs0 0) faces S.empty S.empty []
  where
    gs0 = GenerationState (kGen initBC, N, initBC)

bfs :: (BandagedCube -> Int) -> Word8
      -> PS.PSQ#(GenerationState, Word8) -> [Face]
      -> SetVisitedKeys -> SetVisitedKeys -> [(Int, Word8)]
      -> [(Int, Word8)]

bfs kGen maxDepth pq faces visited onceEnqueued acc
  | PS.null pq = acc
  | isRepeated = bfs kGen maxDepth pqNoMin faces visited

```

Capítulo B. Código

```
    onceEnqueued acc
| currDepth > maxDepth = acc
| currDepth == maxDepth = bfs kGen maxDepth pqNoMin faces
  nextVSet (S.delete thisKey onceEnqueued) newChanges
| otherwise = bfs kGen maxDepth nextPQ faces nextVSet
  nextEnq newChanges

where
  --Comprobations
  (thisGenState PS.:-> currDepth , pqNoMin) = fromJust
    (PS.minView pq)
  GenerationState (thisKey, _, _) = thisGenState
  isRepeated = S.member thisKey visited

  --Generation of next layer
  infListNextDepth = (repeat (1 + currDepth))
  nextGS = nextLayerNonRepeating kGen thisGenState faces
    visited onceEnqueued
  nextPQ = insertList (zip nextGS infListNextDepth)
    pqNoMin

  nextVSet = S.insert thisKey visited
  newChanges = (thisKey , currDepth) : acc

  keysEnq = map (\(GenerationState(k, _, _)) -> k) nextGS
  nextEnq = S.union (S.delete thisKey onceEnqueued)
    (S.fromList keysEnq)

  insertList :: (Ord k, Ord p) => [(k , p)] -> PSQ k p -> PSQ k p
  insertList [] pq = pq
  insertList ((k , p):xs) pq = PS.insert k p (insertList xs pq)

  nextLayerNonRepeating :: (BandagedCube -> Int)
    -> GenerationState -> [Face]
    -> SetVisitedKeys -> SetVisitedKeys ->
      [GenerationState]

  nextLayerNonRepeating kGen (GenerationState(_, lastFace,
    bCube)) faces visited onceEnqueued = newStatesFiltered
  where
    moves = [ (f, Turn(f, m)) | f <- faces, m <- [1 .. 3],
      (axisOfFace f /= axisOfFace lastFace) || (f >
        lastFace), validTurn bCube f]
    possibleAccessibleStates = [(lf, tryToTurn bCube m) |
      (lf, m) <- moves, isJust (tryToTurn bCube m) ]
    newStatesFiltered = [ GenerationState (kGen bc, lf,
```

```

bc)  |
      (lf, Just bc) <-
          possibleAccesibleStates,
          S.notMember (kGen bc) visited,
          S.notMember (kGen bc) onceEnqueued ]

```

```

lookupCorners :: BandagedCube -> Word8
lookupCorners = lookupPiece 0

```

```

lookupFstEdges :: BandagedCube -> Word8
lookupFstEdges = lookupPiece 1

```

```

lookupSndEdges :: BandagedCube -> Word8
lookupSndEdges = lookupPiece 2

```

```

lookupPiece :: Int -> BandagedCube -> Word8
lookupPiece n bc =
  let (c, e1, e2) = stdVectors
  in case n of
    0 -> (V.!) c (cornersKey bc)
    1 -> (V.!) e1 (edgesKeyFst bc)
    2 -> (V.!) e2 (edgesKeySnd bc)
    _ -> 25

```

B.4.4. Fichero KorfHeuristic.hs

```

module KorfHeuristic(korfHeuristic, korfIndivHeuristics) where

import GenKorfHeuristics
import Bandaged

-- | DEFINITIVE method for estimating the minimal moves
-- remaining at a position
korfHeuristic :: BandagedCube -> Int
korfHeuristic bc = (fromIntegral hDef) :: Int
  where
    (c, e1, e2) = lookupAll bc
    hDef = maximum [c, e1, e2]

korfIndivHeuristics :: BandagedCube -> [Int]
korfIndivHeuristics bc = (map fromIntegral hs) :: [Int]
  where
    (c, e1, e2) = lookupAll bc
    hs = [c, e1, e2]

```

B.5. Módulo Search-Engine

B.5.1. Fichero Search.hs

Capítulo B. Código

```
module Search(genericSearch) where

import Bandaged
import Moves
import Data.Maybe(fromJust)
import Data.List(nub)

-- | SearchingState storages all the information needed to to a
-- Search.
data SearchingState = SearchingState {found :: Bool,
                                         initialState :: BandagedCube,
                                         currentDepth :: Int,
                                         maximumDepth :: Int,
                                         condition :: (BandagedCube -> Bool),
                                         solution :: [Turn],
                                         validLayers :: [Face],
                                         listMoves :: [Turn],
                                         lastFace :: Face,
                                         heuristic :: (BandagedCube -> Int),
                                         minimumExceding :: Int
                                         }

instance Show SearchingState where
  show (SearchingState f ini currD maxD _ sol layers _
        lstFace _) =
    "found: " ++ show f ++ "\n" ++
    "initial state: " ++ show ini ++ "\n" ++
    "current depth: " ++ show currD ++ "\n" ++
    "maximum depth: " ++ show maxD ++ "\n" ++
    "solution: " ++ show sol ++ "\n" ++
    "generation of moves: " ++ show layers ++
    "last face executed: " ++ show lstFace

-- | Recieves data, makes a generic bounded search and compose
-- the solution
genericSearch :: BandagedCube           -- ^ Initial state
              -> (BandagedCube -> Bool)  -- ^ Condition to
                                         determine a Node is found
              -> [Turn]                  -- ^ List of Turns
                                         to generate a new node
              -> (BandagedCube -> Int)   -- ^ Heuristic
                                         (must be admissible)
              -> Maybe Algorithm         -- ^ The solution

genericSearch ini cond validMoves h
  | found search = Just (Algorithm (solution search))
```

```

--Solution found
| otherwise = Nothing
--Solution not found
where
  validLs = nub (map (\(Turn(f,_)) -> f) validMoves)

initialSS = SearchingState{found = False, initialState
  = ini, currentDepth = 0, maximumDepth = h ini,
  condition = cond, solution = [], validLayers =
  validLs, listMoves = validMoves, lastFace = N,
  heuristic = h, minimumExceding = maxBound :: Int}

search = idaStar initialSS

idaStar :: SearchingState -> SearchingState
idaStar initSS
| found thisSearchSS = thisSearchSS
--Update max depth with minimum node that exceeded the max.
| (nextDepth > threshold) = idaStar (initSS {maximumDepth =
  nextDepth})
| otherwise = initSS
where
  thisSearchSS = dfsSgle initSS
  threshold = maximumDepth initSS
  nextDepth = minimumExceding thisSearchSS

-- | Search with dfs from one node
dfsSgle :: SearchingState -- ^ Initial Searching State
  -> SearchingState -- ^ Final Searching State

dfsSgle initialSS
--solution found
| predicate ini = initialSS {found = True}
--pruning, reached maximum depth
| currD > maxD || (currD + h ini > maxD) = prunedSS
--intermediate, keep searching
| otherwise = dfsMult initialSS movesToIterate
where
  (SearchingState _ ini currD maxD predicate --
   movesValid lstFace h exc) = initialSS
  estimLength = currD + h ini

  prunedSS = if ((estimLength > maxD) && (estimLength <
  exc))
  then
    (initialSS{minimumExceding = estimLength})
  else

```

Capítulo B. Código

```
initialSS
movesToIterate = filter (predValidCanonicSequence ini
1stFace) movesValid

predValidCanonicSequence :: BandagedCube -> Face ->
    Turn -> Bool
predValidCanonicSequence bc lsface (Turn(f,_)) =
    ((axisOfFace f /= axisOfFace lsface) || (f >
        lsface)) &&
    (validTurn bc f)

-- | Search with dfs algorithm. Iterate over several move
-- generation
dfsMult :: SearchingState
    -> [Turn]
        turns used to generate branches
    -> SearchingState
        -- ^ Initial
        -- ^ List of
        -- ^ Final

--ended iterating
dfsMult initialSS [] = initialSS
--keep iterations
dfsMult initialSS (x:xs)
--Correct branch, recompose solution
| found thisBrach = thisBrach {solution = (x : solutionP)}
--Incorrect branch, keep searching
| otherwise = dfsMult (initialSS {minimumExceding = min
    exc0 maybeNewExc}) xs
where
    (SearchingState _ ini currD _ _ _ _ _ exc0) =
        initialSS
    nextState = tryToTurn ini x
    (Turn(lastFaceExecuted, _)) = x
    thisBrach = dfssgle (initialSS
        {initialState = fromJust nextState,
        currentDepth = currD + 1,
        lastFace = lastFaceExecuted,
        minimumExceding = exc0
        }
    )

maybeNewExc = minimumExceding thisBrach
solutionP = solution thisBrach
```

B.5.2. Fichero MoveGeneration.hs

```
module MoveGeneration(sixAxis, freeFaces, kociembaMoves,
    notBlockedMoves, numberCanonicalSequences) where
```

```

import Moves
import qualified Data.Set as S
import Bandaged

-- | Returns the moves achievable with 6 faces (R, R', R2, U
-- ...
sixAxis :: [Turn]
sixAxis = casualUnzip (map (casualZip [1 .. 3])) [R .. ] )

-- | Returns the possible moves of Kociemba 2nd step (R2, L2,
-- F2, B2, U, D...)
kociembaMoves :: [Turn]
kociembaMoves = casualUnzip (qt ++ ht)
  where
    qt = map (casualZip [1 .. 3]) [U, D]
    ht = map (casualZip [2]) [R, L, F, B]

-- | Returns the layers that can be moved (not blocked) in a
-- bandaged cube
notBlockedMoves :: BandagedCube -> [Turn]
notBlockedMoves bc = freeFaces (movableFaces bc)

-- | Returns the possible moves given a list of layers
freeFaces :: [Face] -> [Turn]
freeFaces xs = casualUnzip (map (casualZip [1..3])) xs

casualZip :: [Int] -> Face -> (Face, [Int])
casualZip xs f = (f, xs)

casualUnzip :: [(Face, [Int])] -> [Turn]
casualUnzip xs = concat (map indivFace xs)
  where
    indivFace :: (Face, [Int]) -> [Turn]
    indivFace = ( \(f, numsList) -> [Turn (f, n) | n <-
      numsList] )

-- | Returns the faces that can be moved
movableFaces :: BandagedCube -> [Face]
movableFaces bCube = S.toList (S.difference (S.fromList [R, U,
  F, L, D, B]) allBlockedFaces)
  where
    allRestr = restrictions bCube
    allBlockedFaces = S.unions (S.map (facesBlockedByBlock)
      allRestr)

    -- Returns the layers that each block is blocking (when
    -- is over >1 centers)

```

Capítulo B. Código

```
facesBlockedByBlock :: S.Set Int -> S.Set Face
facesBlockedByBlock xs
  | (isBlockingCenters) = S.map (centerToLayer)
    centers
  | otherwise = S.empty
where
  centers = S.filter (>= 48) xs
  isBlockingCenters = (length centers) >= 2

centerToLayer :: Int -> Face
centerToLayer 48 = U
centerToLayer 49 = F
centerToLayer 50 = R
centerToLayer 51 = B
centerToLayer 52 = L
centerToLayer 53 = D
centerToLayer _ = N

numberCanonicalSequences :: Int -> [Face] -> [(Int, Int)]
numberCanonicalSequences n faces = map (\m -> (m, length
  (genAllCanonics m turns))) [1 .. n]
where
  turns = [Turn(f, m) | f <- faces, m <- [1..3]]

genAllCanonics :: Int -> [Turn] -> [[Turn]]
genAllCanonics 1 xs = map (\m -> [m]) xs
genAllCanonics nMoves xs = concat (map (\m ->
  addOneMove m minusOne) xs)
where
  minusOne = genAllCanonics (nMoves-1) xs

addOneMove :: Turn -> [[Turn]] -> [[Turn]]
addOneMove m xs = map (\moves -> (m : moves)) sublist
where
  sublist = filter (\alg -> case alg of
    [] -> canJoinMoves
    m (Turn(N, 0)) ->
    (x:_)
      -> canJoinMoves m x
  ) xs

canJoinMoves :: Turn -> Turn -> Bool
canJoinMoves t1 t2 = (axisOfFace f1 /= axisOfFace f2)
  || f1 < f2
where
  Turn (f1,_) = t1
  Turn (f2,_) = t2
```

B.5.3. Fichero SolvingStrategies.hs

```

module SolvingStrategies(iddfsSolver, kociembaSolver,
    smartKorfSolver) where

import Bandaged(BandagedCube(..), solvedBC, tryToExecuteAlg)
import Moves

import Search
import KorfHeuristic
import MathematicalNotation(edgesState, cornerState)

import MoveGeneration(sixAxis, kociembaMoves, notBlockedMoves)

import Data.Maybe(fromJust, isJust)
import Data.List(sort)

-- | Solves the cube with iddfs algorithm (deprecated in the
-- future)
iddfsSolver :: BandagedCube -> Maybe Algorithm
iddfsSolver bc = genericSearch bc (solvedBC) sixAxis (const 0)

kociembaState :: BandagedCube -> Bool
kociembaState bc = (sumOrientations == 0) && (sort middleEdges
== [4 .. 7])
where
    (_ , co) = cornerState bc
    (ep, eo) = edgesState bc
    middleEdges = ((take 4) . (drop 4)) ep
    sumOrientations = sum co + sum eo

-- | Solves the cube with the Kociemba Algorithm. Might not end
-- depending on the bandages
kociembaSolver :: BandagedCube -> Maybe Algorithm
kociembaSolver bc
    | (isJust algStep1) && (isJust algStep2) = Just ((fromJust
        algStep1) <> (fromJust algStep2))
    | otherwise = Nothing
where
    algStep1 = genericSearch bc (kociembaState) sixAxis
        (const 0)
    bcIntermediate = algStep1 >>= (\algUnpack ->
        tryToExecuteAlg bc algUnpack)
    algStep2 = bcIntermediate >>= (\bcUnpack ->
        genericSearch bcUnpack solvedBC kociembaMoves
        korfHeuristic)

-- | Solves the cubo optimally with the Korf algorithm. Use

```

Capítulo B. Código

```
only the movable faces
smartKorfSolver :: BandagedCube -> Maybe Algorithm
smartKorfSolver bc = genericSearch bc (solvedBC)
  (notBlockedMoves bc) korfHeuristic
```