Repositorio descentralizado para la gestión y el aprendizaje de algoritmos de speedcubing
Trabajo final de carrera

Agustin Villarreal
agustinvilla@frba.utn.edu.ar

Cristobal Szkutnik
crisszkutnik@gmail.com

Guido Dipietro
dipietroguido@gmail.com

Guido Enrique Zabala
guidoenr4@gmail.com

Matias Davicino
mdavicino@frba.utn.edu.ar

Universidad Tecnológica Nacional, Facultad Regional Buenos Aires

Abstract
El presente documento narrará en profundidad el desarrollo de un sistema totalmente descentralizado construido para suplir la necesidad de la existencia de un repositorio de algoritmos de speedcubing [1] que pueda ser mantenido por la propia comunidad que lo utiliza y no por una entidad única. De la misma forma que los ajedrecistas estudian aperturas, los speedcubers estudian algoritmos para resolver cubos de Rubik de formas más rápidas.

Palabras Clave
descentralización, blockchain, web3, speedcubing, cubo de rubik

Introducción

El sistema desarrollado se encuentra estrictamente ligado a una problemática presente en el mundo del speedcubing. De la misma manera que un ajedrecista necesita estudiar aperturas, analizarlas, y practicarlas, un speedcuber necesita aprender los denominados "algoritmos".

Se denomina de esta forma en el speedcubing a una secuencia de movimientos que, al aplicarse en un puzzle, afectan a las piezas del mismo de una forma previamente conocida. Es decir, un algoritmo se utiliza para resolver problemas comunes de manera óptima, previa memorización del mismo.

Hasta el desarrollo de este trabajo, no existía ningún tipo de sistema o plataforma que reuniera todos los algoritmos utilizados por los métodos más comunes en un mismo lugar, donde además se pudiera practicarlos siguiendo el progreso personal, y no se dependiera de una única persona o entidad para mantenerlo. En reiteradas ocasiones existieron proyectos similares, pero al siempre depender de una única persona resultaron insostenibles en el tiempo y la comunidad de speedcubing se encontró migrando una y otra vez de plataforma.

[1] El speedcubing es el deporte de resolver cubos de Rubik o puzzles similares en el menor tiempo posible. Quienes lo practican se conocen como "speedcubers".

Se busca que *Cubitorium*[2] sea la alternativa final para los speedcubers en este sentido. Para proveer descentralización y trasladar el mantenimiento del sistema a la comunidad misma se desarrolló un back-end hosteado en la blockchain de Solana[1], y un front-end desplegado utilizando la tecnología de IPFS[2].

En este documento se narrará en detalle la naturaleza de un algoritmo de speedcubing, así como la implementación paso a paso de cómo se ha modelado un cubo de Rubik programáticamente para poder aplicar secuencias de movimientos y validar que funcionen correctamente, y la arquitectura del sistema en cuestión.

Speedcubing

Se narrará primeramente el funcionamiento de un método de speedcubing a grandes rasgos, y el uso esperado de este sistema por parte de un speedcuber.

En este deporte, cada puzzle se resuelve de acuerdo a un método que divide el problema total en distintas etapas o sub-problemas. Algunas de ellas, generalmente las iniciales, se pueden completar de forma intuitiva. Sin embargo, al avanzar en la resolución, el rompecabezas se restringe hasta llegar a un punto en el que es extremadamente difícil deducir soluciones en un tiempo muy corto. Por esta razón, las últimas etapas de los métodos de speedcubing consisten en aplicar un algoritmo que resuelva el caso que se presenta.

Si un método tiene una última etapa con 22 casos posibles[3], entonces un speedcuber memorizará y practicará todos estos casos para poder resolver cualquier estado que se le presente al llegar a esta etapa en segundos.

De este modo, una vista del sistema podría verse como sigue:

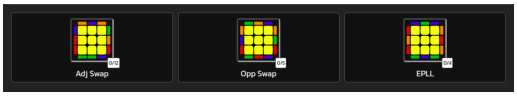

Figura 1: Vista de etapa “PLL” en SpeedcubeDB[3]

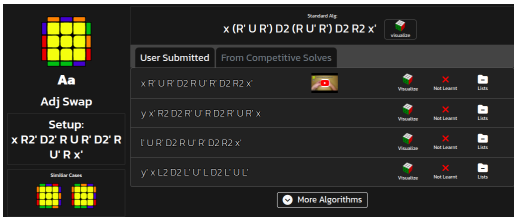

Figura 2: Vista de algoritmo puntual en SpeedcubeDB[3]

Allí, un usuario podría elegir una serie de algoritmos del “catálogo” y en base a eso leerlos, aprenderlos, y practicarlos.
Los algoritmos no son únicos. Es decir, para cada caso existen varias formas (de hecho, infinitas) de resolverlo. Sin embargo, solo algunas de estas soluciones son viables para el speedcubing. Muchas de ellas son igualmente buenas, y la preferencia sobre una u otra es una cuestión meramente personal. En algunos casos, incluso, sucede que un speedcuber descubre una nueva forma de resolver un caso y causa que sea la nueva opción preferida.

Modelo matemático

Dada la realidad enunciada en la sección anterior, nuestro sistema admite la carga de algoritmos nuevos. Al carecer de usuarios privilegiados[4], la única manera de validar que el algoritmo ingresado sea correcto es implementando un modelo matemático del cubo de Rubik que permita validar las soluciones automáticamente.

[2]Nombre comercial del sistema.

[3]La última etapa del método más popular para el cubo de Rubik, “PLL”, tiene 22 casos posibles incluyendo el estado resuelto.

[4]El sistema cuenta con usuarios administradores, pero no se espera que formen parte de los casos de uso típicos del mismo. Es decir, estos usuarios solamente se encargarían de definir configuraciones globales del entorno por única vez o muy rara vez.

Narraremos a continuación cómo se ha conseguido hacer eso exitosamente.

Un cubo de Rubik puede modelarse muy sencillamente como dos arreglos de un tipo abstracto. Al tener un cubo 12 aristas y 8 esquinas[5], bastaría con modelar dos tipos de elementos y contar con 12 y 8 instancias de cada uno respectivamente.

En pseudocódigo:

```
pieza {
    id: T,
    orientacion: u8
}

esquinas: [pieza; 8]
aristas: [pieza; 12]
```

En esta representación, llamamos "*id*" a un identificador unívoco para cada pieza. Por ejemplo, podría tratarse de los colores que la componen, o un número arbitrario de referencia. Por otro lado, llamamos "*orientación*" a la dirección en la que se encuentre esta pieza. Debido a que las esquinas tienen 3 lados y las aristas 2, puede suceder que se encuentren en la misma posición pero con una orientación diferente.

Dada la posición relativa de cada elemento dentro de su arreglo, podremos determinar su permutación. Dado el valor que contenga, podremos determinar su orientación.

Esta representación es una de las posibles, y la usaremos de base para modelar el cubo de Rubik en el back-end de Solana programado en Rust[4].

En nuestro sistema, en lugar de contar con arreglos de una estructura, decidimos separarlo en cuatro arreglos por mera conveniencia: uno para la orientación y uno para la permutación de cada tipo de pieza, dando 4 en total:

```
struct Cube {
    co: [u8; 8],
    cp: [u8; 8],
    eo: [u8; 12],
    ep: [u8; 12],
}
```

Estos arreglos funcionarán como dos grupos de dos, siendo que la permutación de los elementos se correspondan a aquellos arreglos modelando tipos distintos de piezas debe ser modificada siempre de la misma forma.

Usaremos los prefijos "o" y "p" para referirnos a la orientación y permutación, y los sufijos "c" y "e" para referirnos a las esquinas (*corners*) y aristas (*edges*).

Como identificadores, utilizaremos números enteros. Es decir, `[1,4,2,3,6,5,8,7]` podría representar una permutación válida de las esquinas (`cp`), mientras que `[1,2,3,4,5,6,7,8]` representaría el estado resuelto.

Para la orientación, definiremos de forma arbitraria valores para cada tipo de pieza según su característica. Esto significa valores posibles de `0, 1 y 2` para las esquinas (al tener 3 lados), y de `0 y 1` para las aristas por la misma razón. Definiremos la orientación de las esquinas como se muestra, según la ubicación del "sticker" blanco o amarillo de cada una:

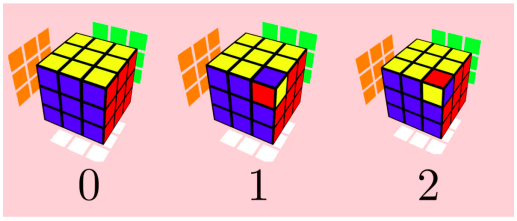

Figura 3: Definición de la orientación de esquinas

De forma similar, definiremos la orientación de las aristas como una propiedad intrínseca de cada pieza que cambia de valor cada vez que esa pieza está presente en la cara frontal o trasera y esa misma cara rea-

[5]En la jerga del speedcubing se denomina "esquina" a un vértice del cubo.

liza un giro simple.[6] Con esto en mente, `[0,1,1,0,0,1,1,0,0,1,1,0]` podría ser un arreglo de orientaciones de aristas válido (`eo`), y `[0,0,0,0,0,0,0,0,0,0,0,0,0,0]` el estado inicial del mismo.

Dadas estas definiciones, podremos construir un cubo de Rubik resuelto como sigue:

```
let cube: Cube = Cube {
   cp: Vec::from_iter(1..=8),
   co: vec![0; 8],
   ep: Vec::from_iter(1..=12),
   eo: vec![0; 12],
};
```

Finalmente, solo restará definir cómo modifica cada giro aplicado en el mismo a estos arreglos, y con esto podremos aplicar movimientos en nuestro cubo de Rubik modelado matemáticamente.

Arquitectura del sistema y tecnologías

Esta lógica se implementa en Rust[4], en un programa de Solana[1] desarrollado bajo el framework de Anchor[5]. El código fuente aún no ha sido publicado al momento de la redacción de este documento.

Al contar con un programa de Solana, se resuelven dos componentes principales del sistema: el procesamiento en back-end y la persistencia. La blockchain de Solana hace las veces de ambos componentes:

- **Persistencia:** Solana es una blockchain con alta capacidad de persistencia escalable. Cada programa puede reservar dinámicamente el almacenamiento que precise y persistir allí los datos de manera indefinida.[7]
- **Procesamiento:** Los programas en Solana existen con el propósito de realizar cálculos de manera descentralizada. Este componente reemplaza lo que, de haber implementado este sistema de forma tradicional, podría haber sido un módulo de *cloud computing*. Solana permite un *throughput* de 65000 transacciones por segundo[6].

Descentralizar esta parte del sistema de esta manera nos permite desentendernos de los principales factores que funcionaban como dolor en los otros sistemas que buscaron resolver la problemática que atacamos. Por un lado, una vez desplegado el programa en Solana ya no hay que encargarse de realizar pagos periódicos o de ningún tipo para mantenerlo en funcionamiento. Por el otro, nadie podrá darlo de baja una vez sea publicado.

Seguirá existiendo la necesidad de fondear el sistema esporádicamente, ya que aumentar el espacio de almacenamiento[8] implica tener que pagar por cada byte adicional, y esta responsabilidad fue trasladada al programa que será fondeado inicialmente. Será en el futuro la misma comunidad de speedcubing la encargada de transferir dinero a la cuenta del programa.[9]

Existe una capa intermedia entre el back-end (programa blockchain) y el front-end: el SDK. Este módulo, materializado como una clase de TypeScript[8] provee encapsulamiento de todas las funcionalidades englobadas por el programa.

Ya que los programas en Solana solo admiten comunicaciones en formato serializado, es necesario abstraer este comportamiento.

[6] Esta decisión es arbitraria en cuanto al eje, pero tiene una explicación matemática por detrás relacionada a la teoría de grupos. Consideramos que excede el objetivo de este documento, por lo que no se desarrollará su naturaleza.

[7] El almacenamiento tiene un costo proporcional a los bytes almacenados, pero este costo es muy bajo (0.00089784 SOL por byte al momento de escribir este documento, cerca de 2 centavos de dólar).

[8] Precisamente, cada *alloc* de *Program Derived Accounts*.

[9] La cuenta donde se almacenan las criptomonedas no tiene custodia; es decir, la única forma de usar ese dinero es mediante la lógica misma que está redactada en ese programa. Una vez enviada ahí, nadie puede usarla para ningún otro propósito que no esté escrito en su código.

La estructura de un llamado a un programa en Solana es la siguiente:

```
struct ProgramCall {
    program_id: Pubkey,
    data: &[u8],
    accounts: &[AccountInfo],
}
```

En donde *program_id* es la dirección o identificador del programa, *data* es el payload serializado[10], y *accounts* es un arreglo de las cuentas de contexto que utilizará la instrucción[11].

Por lo tanto, todo llamado al programa debe contener un indicador que referencie a qué función llamar, seguido de eventuales argumentos. Es precisamente esto uno de los problemas que el framework Anchor[5] nos resuelve. Anchor asigna a cada función (denominadas "instrucciones") un hash unívoco[12], y luego serializa según el estándar de *borsh*[9] los parámetros enviados.

Aún así, si bien esto se materializa como un cliente de Typescript provisto por Anchor, no es suficiente para utilizar desde un front-end. Sucede que el tercer argumento, *accounts*, no siempre se completa de forma trivial[13]. Es por esto que mediante una clase personalizada es posible encapsular la lógica necesaria para llenar este campo y abstraer al desarrollador front-end de las particularidades de la blockchain de Solana a la hora de integrar su desarrollo con el módulo de back-end provisto.

Finalmente, el sistema tiene una interfaz gráfica para el usuario desarrollada en React[7]. Este módulo interactúa de forma directa con el SDK descripto anteriormente, quien a su vez se comunica directamente con la blockchain de Solana. Para dar completitud a la descentralización del sistema, todo el front-end se despliega en el sistema de IPFS[2].

IPFS es un filesystem distribuido que direcciona los archivos en base a su contenido, en lugar de en base a su dirección como lo hace la web tradicional. Los archivos subidos a este sistema serán replicados en los nodos de la red y, al momento en el que un usuario desee acceder al archivo, será accedido del nodo más cercano de una forma similar a la que un sistema *torrent* lo consigue.

Los nodos de IPFS son ligeros y sencillos de levantar[14], pero también son sencillos de dar de baja. Si el archivo en cuestión no fue transmitido lo suficiente, desaparecerá por siempre de la red. Para mitigar este problema, existen servicios de "pinning" que activamente mantienen vivo un archivo con el fin de que no desaparezca de la red.

Un archivo accedido frecuentemente por una comunidad entera globalizada difícilmente será eliminado de la red, ya que éste será replicado en numerosos nodos distribuidos de una manera geográficamente diversa.

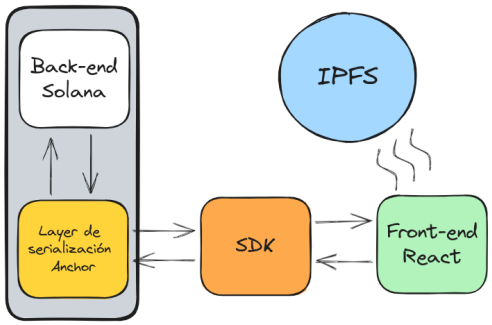

Figura 4: Arquitectura del sistema

[10] En Rust, `&[u8]` es comparable a un `void*` en C.

[11] Resumidamente, las direcciones de memoria solo lectura y modo escritura que accederá una función durante su ejecución.

[12] Primeros 8 bytes del hash SHA256 de la cadena ``global:x'' siendo x el nombre de la instrucción.

[13] Resumidamente, algunas direcciones de memoria se deben calcular o averiguar realizando consultas de antemano en Solana.

[14] De hecho, el navegador Brave funciona como un nodo de IPFS[10].

Conclusión y Trabajos Futuros

Se concluye la importancia de la independencia de una comunidad como grupo. Siendo un conjunto muy grande, diverso y distribuido de personas con un interés en común, es menester ser capaz de entregar la custodia y responsabilidad de mantenimiento de los servicios utilizados a la misma comunidad, sin depender de una parte externa.

En el pasado la comunidad de speedcubing ha sufrido muchos inconvenientes por el hecho de no contar con un repositorio de algoritmos descentralizado, siendo el último de estos retrocesos este mismo año, en el que SpeedcubeDB ha sido dado de bajas numerosas veces.

La tecnología de la era que vivimos ya nos entrega la capacidad de construir sistemas con paradigmas totalmente diferentes a lo habitual, a modo de empoderar a un grupo de personas a través de la lógica de un programa y no depender simplemente de la confianza y la buena voluntad de partes desconocidas.

Como trabajos futuros se proponen extensiones de *Cubitorium* para almacenar otro tipo de informaciones útiles para speedcubers, como reconstrucciones o distintos métodos, a la par de que se busca incentivar el desarrollo de nuevos sistemas descentralizados en contextos en los cuales sean una solución apropiada, más allá del speedcubing.

Referencias

[1] Blockchain de Solana. `https://solana.com/`.

[2] InterPlanetary File System. `https://ipfs.tech/`.

[3] SpeedcubeDB. Repositorio similar a Cubitorium que será descontinuado pronto. `https://speedcubedb.com/`.

[4] Lenguaje Rust. `https://www.rust-lang.org/`.

[5] Anchor, framework de desarrollo de programas de Solana. `https://www.anchor-lang.com/`.

[6] Estudio del rendimiento de Solana en octubre del 2022. `https://solana.com/news/network-performance-report-october-2022`.

[7] React, librería para desarrollo front-end. `https://react.dev/`.

[8] Lenguaje Typescript. `https://www.typescriptlang.org/`.

[9] Borsh: Binary Object Representation Serializer for Hashing. `https://borsh.io/`.

[10] Soporte de IPFS en navegador Brave. `https://brave.com/ipfs-support/`.