CIFRAS, LETRAS Y ALGORITMOS

TRABAJO FIN DE GRADO
CURSO 2024-2025

AUTOR
LUIS REBOLLO CÓCERA

DIRECTOR
RUBÉN RAFAEL RUBIO CUÉLLAR

GRADO EN INGENIERÍA DE SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TÍTULO
CIFRAS, LETRAS Y ALGORITMOS

TRABAJO DE FIN DE GRADO EN INGENIERÍA DE SOFTWARE

AUTOR
LUIS REBOLLO CÓCERA

DIRECTOR
RUBÉN RAFAEL RUBIO CUÉLLAR

CONVOCATORIA: SEPTIEMBRE
CALIFICACIÓN: 7,8

GRADO EN INGENIERÍA DE SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

1 DE SEPTIEMBRE DE 2025

RESUMEN

CIFRAS, LETRAS Y ALGORITMOS

El concurso televisivo *Cifras y letras*, que ha vuelto recientemente a la parrilla de TVE, propone una serie de pruebas que desafían la agilidad mental, el dominio del lenguaje y la capacidad de cálculo de los participantes. Entre ellas, destacan dos que dan nombre al programa: *La cifra exacta*, que consiste en alcanzar un número objetivo combinando operaciones aritméticas básicas a partir de seis cifras dadas y *La palabra más larga* que reta a formar, a partir de diez letras concretas, la palabra más extensa posible contenida en el vocabulario del idioma español.

Este proyecto aborda la resolución automatizada de ambas pruebas, mediante el uso de algoritmos de búsqueda, estructuras de datos y técnicas propias de la ingeniería del software. El objetivo es estudiar el comportamiento y el rendimiento de diversas implementaciones alternativas y relacionarlas con soluciones existentes.

La herramienta que resuelve las pruebas del concurso se puede ejecutar en cualquier navegador mediante una interfaz web, que interactúa con la implementación de los algoritmos en C++ compilada a WebAssembly.

Palabras clave

Cifras y letras, BFS (búsqueda en anchura), DFS (búsqueda en profundidad), WebAssembly, Árbol de prefijos.

ABSTRACT

CIFRAS, LETRAS Y ALGORITMOS

The television game show *Cifras y Letras* (*Countdown*, in its English version), which has recently returned to the TVE lineup, features a series of challenges that test participants' mental agility, language skills, and reasoning abilities. Notably, two main modalities stand out, giving the show its name: *The Exact Number*, which consists of reaching a target number by combining basic arithmetic operations using six given numbers and the results derived from their combination; and *The Longest Word*, which involves creating the longest possible word in Spanish using ten specific letters.

This project addresses the automated resolution of both challenges through the use of search algorithms, data structures, and software engineering techniques. The objective is to study the behavior and performance of various alternative implementations and relate them to existing solutions.

The tool that solves the contest problems can be run in any browser through a web interface, which interacts with the implementation of the algorithms in C++ compiled to WebAssembly.

Keywords

Countdown, BFS (Breadth first search), DFS (Depth first search), WebAssembly, prefix tree (trie).

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 - Introducción

1.1 Antecedentes y reglas del juego

El programa *Le Mot le plus long* debutó en la televisión francesa en 1965 como un concurso de vocabulario centrado en encontrar la palabra más larga a partir de un conjunto limitado de letras. En 1972 se le añadió una segunda prueba centrada en el cálculo mental, y el programa adoptó su nombre definitivo: *Des chiffres et des lettres* [1]. Desde entonces, ha sido emitido casi ininterrumpidamente en Francia durante más de cincuenta años, y ha sido adaptado en otros países, como en Reino Unido (*Countdown*, desde 1982) o España, donde se estrenó en 1991. En nuestro país ha tenido varias etapas y actualmente ha regresado a la parrilla de TVE.

A lo largo de su historia, el concurso ha mantenido como núcleo dos pruebas fundamentales, *la palabra más larga* y *la cifra exacta*, pilares que han permanecido constantes prácticamente desde sus inicios. No obstante, en determinadas etapas y versiones, el programa ha incorporado otras pruebas complementarias, como retos de cálculo mental rápido, desafíos de vocabulario temático o incluso minijuegos lingüísticos como la palabra oculta o la definición cruzada, con el fin de enriquecer el formato y adaptarlo a nuevas audiencias. Estas pruebas adicionales solían ocupar una sección menor del programa y variaban en función de la edición o del país en el que se emitía.

Este trabajo se centra exclusivamente en las dos pruebas clásicas y más representativas del concurso, abordándolas desde una perspectiva algorítmica y computacional, con el objetivo de automatizar su resolución y analizar las estrategias más eficientes para afrontar los desafíos que plantean desde el punto de vista del desarrollo de software.

En la prueba de la cifra exacta se seleccionan seis números, extraídos aleatoriamente de dos grupos: los llamados "números pequeños" (valores del 1 al 10, pudiendo repetirse) y los "números grandes" (25, 50, 75, 100). A partir de esos seis valores iniciales, el objetivo es construir, mediante operaciones aritméticas básicas (+, −, ×, ÷), una expresión que dé como resultado un número objetivo aleatorio entre 100 y 999.

Cada número se puede usar tantas veces como aparezca. Los resultados intermedios también pueden combinarse, pero solo se permiten operaciones que produzcan resultados enteros y positivos.

El jugador dispone de un tiempo limitado (entre 30 y 45 segundos, según la versión) para hallar una expresión válida. Si ninguno de los concursantes encuentra la solución exacta, se acepta la solución más cercana al objetivo.

Si tenemos como cifras disponibles: {3,4,5,6,6,25} y tratamos de alcanzar el numero

objetivo 597, la mejor solución sería:

Operaciones:	Cifras disponibles:
25×4= 100	{3,5,6,6,100}
100×6=600	{3,5,6,600}
600-3=597	{5,6,597}

solución: ((25 × 4) × 6) − 3 = 597

En la prueba de la palabra más larga los concursantes eligen al inicio cuántas vocales desean obtener dentro del conjunto de diez letras sorteadas, con la condición de que esa cantidad se encuentre entre un mínimo de tres y un máximo de seis. El resto de letras hasta completar las diez se toma de las consonantes. Como en la versión original, las letras pueden repetirse, pero únicamente pueden utilizarse tantas veces como aparezcan en el sorteo. A partir de esas letras, el jugador debe formar la palabra más larga posible reconocida por el diccionario del idioma en el que se juega. En el caso del español, se utiliza el diccionario de la Real Academia Española (RAE) [2], y se aceptan también plurales y formas verbales.

Por ejemplo, a partir del conjunto de letras {U, Q, W, Q, H, U, R, M, O, E} se considerarían válidas palabras como *quorum*, *humero* o *húmero*, independientemente de que el conjunto inicial no incluya signos diacríticos pero dichos signos estén presentes en la palabra final.

1.2 Motivación

El proyecto nace del interés por abordar los desafíos que plantean las pruebas de *Cifras y letras* [3], *que* combinan lógica y lenguaje de una manera atractiva y han dado lugar a múltiples implementaciones informáticas a lo largo del tiempo, tanto en versiones de escritorio como en aplicaciones web. Entre ellas pueden citarse CaLao [4] o Compteurs de mots [5] que reproducen las reglas del concurso y permiten resolver automáticamente las dos pruebas principales.

El propósito del trabajo no es tanto descubrir una solución inédita, sino aplicar una perspectiva rigurosa desde la Ingeniería del Software para diseñar, comparar y perfeccionar distintas estrategias de resolución, estudiando aspectos como su eficiencia, la extensibilidad del sistema y la claridad en la representación de los resultados.

El propósito de este trabajo no es descubrir una solución inédita, sino aprovechar la oportunidad de estudiar y comparar distintas estrategias de resolución desde una perspectiva de Ingeniería del Software. En la prueba de la cifra exacta, el interés se centra en aplicar y confrontar técnicas de búsqueda como BFS y DFS [6], observando su impacto en el tiempo de ejecución y en el consumo de recursos. En la prueba de la palabra más larga, el foco está en el uso de estructuras de datos como el trie [7] y en la incorporación de heurísticas que orienten la exploración hacia soluciones más prometedoras.

De este modo, el proyecto combina un componente técnico de diseño, implementación y análisis de algoritmos con una dimensión cultural y lúdica, al tomar como punto de partida un concurso televisivo muy conocido. En definitiva, un ejercicio formativo que permite aplicar de manera práctica los conocimientos adquiridos durante el grado y, al mismo tiempo, experimentar con enfoques distintos para resolver problemas combinatorios clásicos.

1.3 Objetivos

El principal objetivo de este proyecto es abordar, desde una perspectiva algorítmica las dos pruebas fundamentales del concurso Cifras y letras. El objetivo general es desarrollar implementaciones que permitan resolver estos desafíos de forma eficiente, explorando para ello diversas técnicas aprendidas durante el grado en ingeniería de software.

En primer lugar, se plantea como objetivo comparar distintas estrategias de resolución aplicadas a las pruebas de *La cifra exacta* y *La palabra más larga*. En el caso de *La cifra exacta*, se analizan y experimentan aproximaciones basadas en algoritmos clásicos de búsqueda, como la búsqueda en anchura (BFS) y la búsqueda en profundidad (DFS). La intención es identificar cuál ofrece mejores resultados basándonos en sus tiempos de exploración, la profundidad alcanzada y el número de nodos recorridos.

Otro objetivo paralelo es la implementación de una solución a la prueba de letras. Para ello, nos apoyamos en estructuras de datos como los árboles de prefijos (*trie*). A través de este diseño, se pretende ofrecer una solución capaz de resolver el problema en un tiempo razonable a pesar del abultado vocabulario castellano.

Además de los objetivos algorítmicos, el proyecto incluye una interfaz accesible desde el navegador. Esta página permite resolver las dos pruebas y jugar con casos generados aleatoriamente o introducidos por el propio usuario. La interfaz muestra la solución encontrada por los algoritmos y facilita la comparación con la propuesta del usuario, por ejemplo, contrastando la longitud de la palabra o el número alcanzado. El objetivo es que cualquier persona, sin conocimientos técnicos, pueda probar el sistema, interactuar con él y evaluar sus resultados de forma sencilla.

Finalmente, se aspira a que este trabajo sirva para afianzar habilidades adquiridas durante el grado, abarcando desde la estructuración del código, hasta la aplicación de estructuras de datos y patrones clásicos en la resolución de problemas. El proyecto se concibe como una oportunidad para aplicar dichos aprendizajes en un entorno de producción completo, con resultados medibles y posibilidades reales de mejora y expansión.

1.4 Plan de trabajo

El proyecto se ha abordado mediante un enfoque incremental, en el que cada avance ha sido integrado en el sistema general y evaluado en conjunto. Esta estrategia ha permitido identificar errores de forma temprana, comparar enfoques diversos y ajustar progresivamente la arquitectura del sistema en función de los resultados obtenidos.

Las principales etapas del desarrollo han sido:

- Fase 1: Estudio del problema y definición formal de las reglas. Se realizó un análisis detallado de las dos pruebas principales del concurso: *La Cifra Exacta* y *La palabra más larga*, estableciendo los requisitos técnicos y las restricciones específicas de cada una.
- Fase 2: Diseño de algoritmos de resolución para la prueba de cifras. Se aplicaron algoritmos clásicos de búsqueda como DFS y BFS, analizando su comportamiento y eficacia en la resolución del problema.
- Fase 3: Construcción de una estructura eficiente para la prueba de letras. Se adaptó e implementó una estructura basada en árboles de prefijos (tries), sobre la que se aplicaron diferentes recorridos, podas y heurísticas para comparar la eficiencia en la generación y validación de palabras.
- Fase 4: Desarrollo de una interfaz web básica. Se diseñó una interfaz accesible y funcional utilizando HTML, CSS, Bootstrap [8] y JavaScript, permitiendo ejecutar ambas pruebas desde el navegador y visualizar los resultados de forma clara.
- Fase 5: Integración del motor C++ con la interfaz web. A través de la herramienta Emscripten [9], se compiló el código C++ a WebAssembly [10], permitiendo que el sistema completo pudiera ejecutarse en el propio navegador del usuario.
- Fase 6: Como posible línea de ampliación, se evaluó la incorporación de una tercera prueba inspirada en técnicas de Procesamiento del Lenguaje Natural (PLN). La propuesta consistía en utilizar representaciones vectoriales (embeddings semánticos) para comparar definiciones. Esta aproximación se exploró de forma

experimental, pero se decidió finalmente descartarla al no ajustarse plenamente a los objetivos ni ofrecer resultados satisfactorios dentro del marco del proyecto.

- Fase 7: Integración final y pruebas. Se llevaron a cabo pruebas de conjunto para verificar la estabilidad del sistema, comprobar la coherencia de las soluciones y validar el rendimiento de los distintos algoritmos en situaciones reales.

Durante el desarrollo se emplearon diferentes entornos y herramientas de trabajo, como Visual Studio 2022 para la programación en C++ y Visual Studio Code para la edición de los componentes web y la integración con JavaScript y WebAssembly. Para el control de versiones se utilizó GitHub [11]

La comunicación con el tutor se mantuvo de forma constante, tanto en sesiones presenciales periódicas como a través del correo de la universidad, permitiendo resolver dudas, contrastar enfoques y consolidar las decisiones técnicas adoptadas.

1.5 Estructura de la memoria

Esta memoria se organiza en cinco capítulos, estructurados de forma que reflejan el proceso de desarrollo completo del proyecto, desde su contexto inicial hasta la evaluación de los resultados obtenidos.

- Capítulo 1 introduce el concurso *Cifras y Letras*, sus reglas fundamentales, su historia y contexto televisivo. Se detallan los objetivos del trabajo, la motivación personal y técnica, y la metodología seguida durante el desarrollo.

- Capítulo 2 presenta el estado de la cuestión, incluyendo una revisión de soluciones previas a ambos problemas, tanto desde el punto de vista algorítmico como de software existente. También se describen brevemente las tecnologías utilizadas en el desarrollo, como WebAssembly, Emscripten, JavaScript, CSS o Bootstrap.

- Capítulo 3 aborda en profundidad el desarrollo técnico del sistema. Se explican las ideas generales detrás de los algoritmos implementados y se analizan

formalmente las estructuras subyacentes (como árboles o grafos de estados), la generación de sucesores y las estrategias de búsqueda empleadas. Se distinguen claramente las soluciones para cada prueba del concurso.

- Capítulo 4 está dedicado a la evaluación de los resultados. Se comparan diferentes enfoques y heurísticas, se analizan métricas de eficiencia y complejidad, y se presentan ejemplos concretos que ilustran el rendimiento de los algoritmos implementados.

- Capítulo 5 consta de una presentación de la interfaz visual acompañada de una comparación con el programa televisivo.

- Capítulo 6 recoge las conclusiones generales del trabajo, así como posibles líneas de mejora y ampliación futura.

Tanto el repositorio de código como la versión pública del prototipo están disponibles en:

- Repositorio del proyecto: https://github.com/Luisrebo/CifrasYLetras
- Prototipo: https: //luisrebo.github.io/CifrasYLetras

Capítulo 2 - Estado de la cuestión

El desarrollo de este proyecto se lleva a cabo mediante técnicas y herramientas contrastadas. El propósito de este capítulo es presentar el contexto existente sobre el cual se construyen las soluciones implementadas. Por un lado, los algoritmos y estructuras de datos que sirven como base para la resolución de las pruebas. Por otro lado, las tecnologías y entornos que hacen posible su ejecución en un navegador web.

El proyecto no es el primer programa de este tipo; los primeros se distribuyeron en dos disquetes de 360 KB. También existen algunas páginas web que, aunque de forma sencilla, resuelven las pruebas en distintos idiomas. La mayoría de ellas generan los datos de entrada mediante un proceso pseudoaleatorio que, sin embargo, resulta difícil de diferenciar de una implementación basada en una batería de casos ya resueltos, especialmente al no permitir configurar los datos de entrada de manera personalizada.

En la literatura académica también se han estudiado de manera formal las pruebas de Cifras y Letras. Cabe destacar el trabajo de Alliot y Vanaret [12], que realizaron un análisis muy completo de la prueba de cifras. En su trabajo revisan los algoritmos más habituales para resolverla, como la búsqueda en profundidad (DFS) y la búsqueda en anchura (BFS), y muestran cómo introducir mejoras como el uso de tablas hash permitiendo acelerar la resolución de forma notable. Además, plantean posibles variantes del juego que lo harían más complejo, como aumentar el número de cifras disponibles o permitir operaciones adicionales.

Otro estudio de referencia es el de Simon Colton [13], que aborda el problema desde otra perspectiva. En lugar de centrarse únicamente en la velocidad de resolución, utilizaron el sistema HR3, una herramienta de inteligencia artificial capaz de explorar automáticamente todas las instancias posibles del juego de cifras. Con este enfoque pudieron analizar el espacio completo de problemas, identificar cuáles son más fáciles o difíciles, y hasta proponer nuevas variantes del concurso.

Junto a estos trabajos académicos, también se encuentran disponibles diversas aplicaciones móviles en las principales tiendas (App Store y Google Play) que permiten

resolver automáticamente las pruebas de Cifras y Letras, proporcionando al usuario soluciones inmediatas [14] [15] [16] [17]. Del mismo modo, existen distintas páginas web que ofrecen servicios similares [18] [19].

2.1 Algoritmos de búsqueda y estructuras de datos

En informática, un *algoritmo de búsqueda* es un procedimiento que recorre un conjunto de estados hasta encontrar aquel que satisface una condición concreta. Podemos encontrar infinidad de aplicaciones para estos algoritmos: la selección del mejor movimiento posible en una partida de ajedrez, estrategias de planificación, la lógica interna de una consulta de información en una base de datos, etc.

El espacio de posibles soluciones se modela mediante un grafo en el que cada nodo representa un estado del problema y cada arista una transición válida entre estados. Los algoritmos de búsqueda en árboles que emplearemos en este proyecto son los siguientes:

- *Búsqueda en anchura (BFS):* del inglés *breadth-first search*. Este algoritmo explora el árbol por niveles, finalizando la búsqueda de todos los estados de la misma profundidad antes de proceder con los que tienen profundidad mayor, lo que garantiza encontrar la solución más corta posible. Para hacer esto factible, es necesario disponer de una estructura que vaya almacenando las búsquedas en curso, lo que aumenta el coste en espacio.
- *Búsqueda en profundidad (DFS):* Depth-first search. Explora cada rama desde el origen hasta el final en búsqueda de soluciones, tomando siempre que sea posible como siguiente estado uno de mayor profundidad.

Además de los algoritmos de búsqueda generales, este proyecto se apoya en una estructura de datos especializada: el trie o árbol de prefijos.

Un trie es un árbol ordenado en el que cada arista está etiquetada con un caracter y cada nodo representa un prefijo de las cadenas almacenadas. A diferencia de otras estructuras de almacenamiento de cadenas, como tablas hash o árboles binarios de búsqueda, el trie organiza los datos de manera que todos los prefijos comunes se

representan mediante el mismo camino desde la raíz. Esto permite consultar si una palabra pertenece al diccionario.

Este tipo de estructura tiene numerosas aplicaciones en informática. Se utiliza, por ejemplo, en los motores de autocompletado y sistemas de sugerencias de texto [20], donde resulta fundamental identificar rápidamente los prefijos comunes. También aparece en técnicas de compresión de datos, al aprovechar la coincidencia de segmentos iniciales para reducir el espacio necesario de almacenamiento. En el ámbito de la recuperación de información, los tries permiten la indexación eficiente de grandes volúmenes de cadenas, facilitando búsquedas rápidas y exactas. Asimismo, desempeñan un papel relevante en el procesamiento del lenguaje natural, donde sirven de base para correctores ortográficos y otros sistemas que requieren verificar o completar palabras de manera inmediata.

En el contexto de este proyecto, el trie resulta especialmente adecuado para la prueba de la palabra más larga. El juego exige construir una palabra válida a partir de un conjunto limitado de letras, y el trie permite garantizar en cada paso que el prefijo actual corresponde a un inicio válido de palabra en el diccionario. De este modo, se evita explorar combinaciones de letras que nunca podrían formar palabras, lo que reduce drásticamente el espacio de búsqueda respecto a una generación ciega de permutaciones.

El coste espacial del trie es mayor que el de una lista simple de palabras, puesto que cada nodo debe almacenar punteros hacia sus hijos, pero esta inversión se compensa con búsquedas mucho más rápidas y con la posibilidad de aplicar estrategias de poda y heurísticas durante la exploración.

2.2 Tecnologías y entornos de ejecución

En los últimos años se han consolidado tecnologías que permiten ejecutar programas de alto rendimiento directamente en un navegador web. Entre ellas destaca *WebAssembly* (Wasm), un estándar abierto diseñado como objetivo de compilación común para lenguajes como C, C++ o Rust. Su principal ventaja es que permite ejecutar

código en el navegador con un rendimiento cercano al nativo, manteniendo al mismo tiempo las garantías de seguridad y aislamiento propias de la web.

Para generar módulos WebAssembly se utilizan compiladores como *Emscripten*, que traduce programas escritos en C o C++ a Wasm y produce además el código JavaScript necesario para su integración con páginas web. De este modo, el mismo código que podría compilarse para un entorno de escritorio se reutiliza en la web sin apenas modificaciones, lo que ha facilitado la aparición de aplicaciones interactivas complejas accesibles desde el navegador.

En la capa de presentación, las tecnologías fundamentales son HTML y CSS, responsables de la estructura y el estilo visual de las páginas. Sobre ellas se apoyan *frameworks* como Bootstrap, que simplifican el diseño de interfaces, adaptadas automáticamente a diferentes dispositivos y resoluciones de pantalla.

Por último, JavaScript juega un papel clave como lenguaje de enlace entre el usuario y la lógica de cálculo. Desde el navegador, permite cargar módulos WebAssembly, invocar funciones compiladas y procesar sus resultados para mostrarlos en la interfaz. Su capacidad para manipular dinámicamente el DOM y coordinar eventos posibilita que aplicaciones basadas en Wasm respondan de forma inmediata a la interacción del usuario, sin necesidad de recargar la página.

Capítulo 3 - Desarrollo y análisis de la solución

La resolución algorítmica de las pruebas de *La cifra exacta* y *La palabra más larga* parte de un principio común: cualquier situación alcanzable en el transcurso del juego puede describirse como un estado. Un estado es una representación formal de toda la información necesaria en un momento concreto para determinar si hemos alcanzado una solución o qué pasos se pueden dar a continuación.

En la prueba de *La cifra exacta*, un estado se describe mediante un multiconjunto de números disponibles. Se denomina multiconjunto porque, a diferencia de un conjunto convencional, puede contener elementos repetidos. Este multiconjunto representa las cifras que aún pueden utilizarse. Además, el estado incluye el valor objetivo a alcanzar y un registro de las operaciones realizadas hasta el momento.

En *La palabra más larga*, un estado se compone de un prefijo (la secuencia de letras seleccionadas hasta ahora) y el multiconjunto de letras restantes que pueden emplearse. El prefijo no es cualquier secuencia arbitraria, debe corresponder a un inicio válido de palabra en el diccionario del idioma para el que se esté resolviendo.

Estos estados se relacionan entre sí formando el espacio de estados, que puede visualizarse como un grafo en el que los nodos corresponden a estados y las aristas representan transiciones válidas. Cada camino en este grafo parte del estado inicial (todas las cifras o letras disponibles) y conduce a un estado final (una solución completa o una situación en la que ya no es posible continuar).

En la prueba de cifras, la expansión de un nodo consiste en elegir dos cifras distintas del multiconjunto y aplicar una de las cuatro operaciones aritméticas permitidas. El resultado sustituye a las cifras usadas, generando un nuevo multiconjunto más pequeño. La cantidad de nodos hijos que produce un estado depende del número de combinaciones de cifras disponibles y del número de operaciones aplicables. Si en el estado actual quedan *n* cifras, el número de combinaciones posibles es $\binom{n}{2}$ y, para cada par, pueden aplicarse hasta cuatro operaciones (aunque algunas se descartan por no dar resultados válidos).

De este modo, con 6 cifras iniciales, el árbol de búsqueda tiene una altura máxima de 5 niveles, pues cada operación reduce en uno el número de cifras activas. En un nivel *d*, con $6 - d$ cifras activas, el número máximo de sucesores de un estado se obtiene multiplicando las posibles parejas de cifras por el número de operaciones aritméticas permitidas (suma, resta, multiplicación y división):

$$S(d) = 4 \cdot \binom{6-d}{2}$$

Si lo desglosamos por niveles, podemos calcular los sucesores máximos por estado en cada uno de ellos:

$$\text{Nivel 0 (6 cifras):} \quad S(0) = 4 \cdot \binom{6}{2} = 4 \cdot 15 = 60$$

$$\text{Nivel 1 (5 cifras):} \quad S(1) = 4 \cdot \binom{5}{2} = 4 \cdot 10 = 40$$

$$\text{Nivel 2 (4 cifras):} \quad S(2) = 4 \cdot \binom{4}{2} = 4 \cdot 6 = 24$$

$$\text{Nivel 3 (3 cifras):} \quad S(3) = 4 \cdot \binom{3}{2} = 4 \cdot 3 = 12$$

$$\text{Nivel 4 (2 cifras):} \quad S(4) = 4 \cdot \binom{2}{2} = 4 \cdot 1 = 4$$

Con estos valores, podemos establecer una cota superior del número total de estados (asumiendo un árbol completo sin podas). El número total de estados se calcula como la suma, sobre todos los niveles, del producto de sucesores posibles hasta dicho nivel:

$$\#\text{estados} \leq \sum_{d=0}^{4} \prod_{i=0}^{d} S(i)$$

Al desglosar cada término, se obtiene:

$$\prod_{i=0}^{0} S(i) = 60$$

$$\prod_{i=0}^{1} S(i) = 60 \cdot 40 = 2.400$$

$$\prod_{i=0}^{2} S(i) = 2.400 \cdot 24 = 57.600$$

$$\prod_{i=0}^{3} S(i) = 57.600 \cdot 12 = 691.200$$

$$\prod_{i=0}^{4} S(i) = 691.200 \cdot 4 = 2.764.800$$

Después de evaluar esta expresión, se obtiene:

$$\#\text{estados} \leq 60 + 2.400 + 57.600 + 691.200 + 2.764.800 = 3.516.060$$

Este valor representa el límite teórico. En la práctica, se generan menos estados porque la búsqueda se detiene al alcanzar el número exacto y muchas operaciones no son válidas. Por ejemplo, en el caso {500,1,1,1,1,1} se alcanzaron 1.335.616 estados y en el caso {501,2,4,8,16,32,64} se alcanzaron 2.530.260 estados.

Conviene precisar, sin embargo, que no se trata de un árbol en sentido estricto de teoría de grafos, puesto que diferentes ramas pueden converger en el mismo estado. Por ejemplo, el multiconjunto {1,2,3,4} puede generar tanto {3,3,4} como {1,2,7}, y ambos conducen a {3,7}. En la implementación desarrollada, estas situaciones se distinguen según la secuencia de operaciones que las ha producido, de modo que se consideran estados distintos. Sin embargo, en la literatura se ha propuesto tratarlos como el mismo estado para evitar exploración redundante. Alliot y Vanaret aplican esta idea mediante el uso de tablas hash, lo que permite detectar estados ya visitados y acelerar notablemente la búsqueda. En su análisis formal, calculan que para *n* = 6 el algoritmo en profundidad requiere entre 656.100 y 2.764.800 operaciones, mientras que el algoritmo en anchura necesita entre 287.331 y 1.144.386 operaciones. Además, muestran experimentalmente que el uso de tablas hash reduce el tiempo de resolución de todas las 13.243 instancias estándar de 160 segundos a solo 26.

En la prueba de letras, el árbol se implementa de forma explícita como un árbol de prefijos (trie). Cada nodo del trie representa un prefijo válido según el diccionario, y cada transición añade una letra disponible. Esto reduce drásticamente el espacio de búsqueda, ya que no se exploran secuencias de letras que no conduzcan a palabras válidas.

Una estrategia alternativa, descrita en el artículo Resolviendo Cifras y Letras [21], propone ordenar alfabéticamente las letras de todas las palabras del diccionario y almacenar este texto ordenado en una tabla junto con la palabra original.

Durante la resolución, según se explica en el propio blog, no basta con usar todas las letras sorteadas, ya que puede ser necesario descartar algunas, al no existir ninguna palabra que las contenga todas. Por ello, se deben considerar todas las combinaciones posibles de letras del sorteo, ordenadas alfabéticamente y comprobar si la clave obtenida se encuentra en la base de datos. El número total de combinaciones a comprobar depende de la longitud máxima L de la secuencia de letras sorteadas y de la longitud mínima k que debe tener una palabra válida. En general, este número se expresa como:

$$\sum_{i=k}^{L} \binom{L}{i}$$

Este valor crece de forma exponencial con el número de letras sorteadas, aunque en el formato real del concurso se mantiene en magnitudes manejables. En el caso del concurso, con $L = 10$ letras y $k = 5$:

$$\sum_{i=5}^{10} \binom{10}{i} = \binom{10}{5} + \binom{10}{6} + \binom{10}{7} + \binom{10}{8} + \binom{10}{9} + \binom{10}{10} = 252 + 210 + 120 + 45 + 10 + 1 = 638$$

Es decir, en cada partida deben evaluarse 638 combinaciones de las diez letras del sorteo como máximo. Como las combinaciones se generan a partir del conjunto inicial de letras previamente ordenado, cada una de ellas conserva ya ese orden y puede utilizarse directamente como clave en la consulta. Cada consulta puede realizarse en tiempo cercano a $O(1)$ si se emplea una estructura de tipo hash.

El preprocesamiento consiste en ordenar las letras de cada una de las palabras del diccionario, lo que se puede hacer con un coste asintótico de $O(L \log L)$ para cada una de las palabras, y almacenar la clave resultante junto con la palabra original en una tabla hash o árbol de búsqueda.

En el caso del trie, la construcción de la estructura se realiza insertando las N palabras del diccionario, cada una de longitud como máximo L. Cada inserción tiene un coste asintótico $O(L)$, por lo que el preprocesamiento completo es $O(L \cdot N)$. El número de nodos creados es también $O(L \cdot N)$, ya que cada nodo corresponde a un prefijo de alguna palabra del diccionario.

Una vez construido, la resolución de la prueba se lleva a cabo mediante un recorrido con backtracking a partir de las letras sorteadas. En el peor caso, este recorrido puede visitar todos los nodos del trie, de modo que su coste se acota igualmente por $O(L \cdot N)$. Sin embargo, en la práctica la disponibilidad de letras reduce drásticamente el número de ramas exploradas: en cuanto no es posible prolongar un prefijo del diccionario con las letras restantes del sorteo, la exploración se interrumpe.

En resumen, el método basado en el trie ayuda a reducir el crecimiento exponencial del número de combinaciones que aparece en el enfoque con tabla hash. Sin embargo, esta diferencia no implica una superioridad teórica absoluta, ya que el coste del planteamiento con el trie depende directamente del tamaño y la estructura del diccionario. En la práctica, las magnitudes del concurso son moderadas y ambos métodos resultan viables. Por ello, la comparación más adecuada entre las dos propuestas debe hacerse de forma experimental.

3.1 Estrategias de búsqueda

Para recorrer estos árboles y localizar soluciones se emplean distintas estrategias:

- Búsqueda en anchura (BFS): recorre el árbol nivel por nivel, garantizando encontrar la solución con menor número de pasos en problemas donde esto tiene sentido (en cifras, la que precisa de menos operaciones). Su coste en

memoria es elevado, ya que requiere almacenar simultáneamente todos los nodos del nivel que está recorriendo.

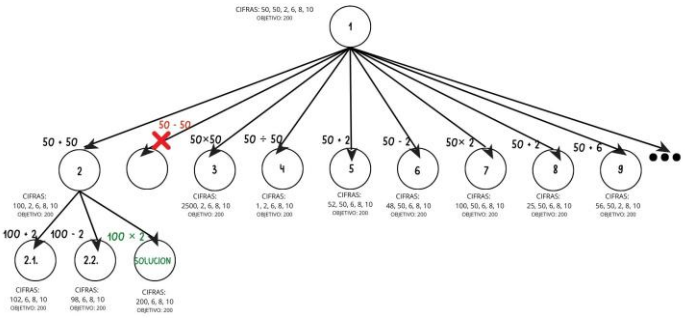

Figura 1. Ejemplo de DFS

- Búsqueda en profundidad (DFS): explora una rama completa antes de retroceder, lo que reduce mucho el uso de memoria. En la prueba de cifras, puede llegar antes a una solución cuando esta se encuentra a gran profundidad, porque BFS tendría que explorar todos los niveles previos antes de alcanzarla. A cambio, no garantiza que la expresión obtenida use el mínimo número de operaciones necesarias.

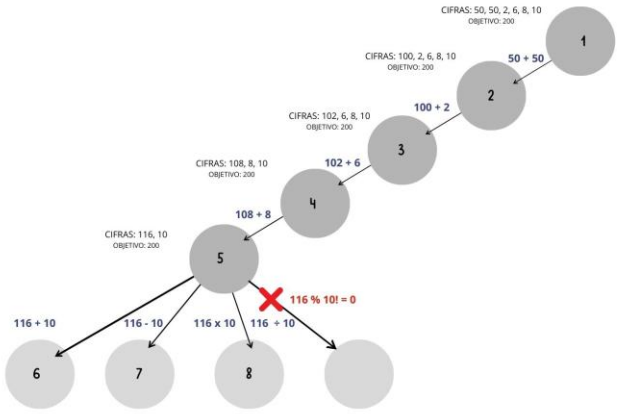

Figura 2. Ejemplo de DFS

- Ramificación y poda: introduce criterios para priorizar ciertos nodos en función de su potencial de llevar a una solución (por ejemplo, en letras, dar prioridad a los nodos con mayor altura, al tener al menos una palabra más larga que el resto).

En la prueba de letras, el recorrido no genera permutaciones explícitas, sino que se apoya en un trie construido a partir del diccionario. Se parte de 10 letras y, en cada paso, se consume exactamente una, de modo que la altura máxima es 10. Aunque un nodo del trie podría tener muchos hijos (uno por cada letra del alfabeto), el factor de ramificación real está acotado por dos condiciones: las letras que todavía quedan disponibles en el multiconjunto (a lo sumo $10 - h$ en un nodo de altura h), y las aristas del trie que efectivamente existen para continuar el prefijo actual en el diccionario. En consecuencia, solo se puede avanzar hacia aquellos hijos que representen extensiones válidas de palabras reales y que respeten las multiplicidades del sorteo. De este modo, cada paso reduce en uno el número de letras restantes y el espacio de búsqueda queda doblemente acotado, por el diccionario y por el conjunto de letras sorteadas, lo que lo hace mucho más manejable que el número de permutaciones posibles.

3.2 Implementación para la cifra exacta

En este apartado se describe cómo se ha implementado la resolución de la prueba de la cifra exacta. Para ello, se definen primero los estados que representan cada situación alcanzable, después se detallan las transiciones válidas entre ellos y se presentan los algoritmos de búsqueda utilizados y las métricas con las que se comparan sus resultados.

3.2.1 Representación de estados

En la prueba de *La cifra exacta*, cada estado se modela como un multiconjunto de números enteros positivos, que representa las cifras disponibles en ese momento, junto con el valor objetivo que se pretende alcanzar. A esto se añaden varios elementos auxiliares que permiten evaluar y reconstruir la solución.

En la implementación, las cifras se almacenan en un array de tamaño fijo que inicialmente contiene todas las cifras disponibles. A medida que se aplican operaciones,

el número de posiciones activas en el array disminuye en función de la profundidad d del estado en el árbol. Tras d operaciones, el total de cifras utilizables es $N - 2d$ donde N es el número inicial de cifras y d es la profundidad del estado.

Cuando se combinan dos cifras mediante una operación válida, el resultado se coloca en la posición de una de las cifras usadas, y en la posición de la otra se inserta el último elemento disponible del array. Con este mecanismo, las cifras activas permanecen siempre en las primeras posiciones, evitando desplazamientos costosos y manteniendo un acceso directo $O(1)$ a cada elemento, véase en la figura 3.

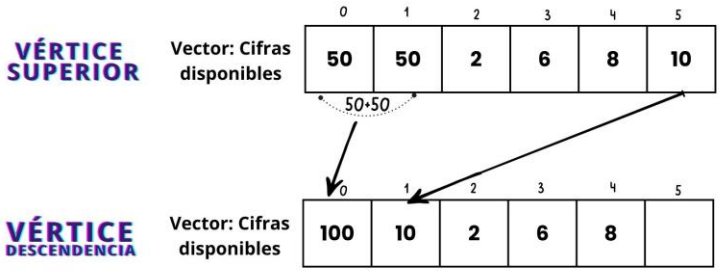

Figura 3. Reubicación cifras en estados descendentes

Además de las cifras activas, cada estado mantiene un array con las cifras utilizadas y otro con los símbolos de las operaciones aplicadas, ambos ordenados según el momento en que se emplearon durante la exploración. De este modo, a las cifras almacenadas en las posiciones $2i$ y $2i + 1$ del array de cifras se les asocia el caracter de la operación i en el array de operaciones. Esta estructura permite reconstruir la secuencia completa de pasos que conducen desde el estado inicial hasta una solución final, ya sea exacta o aproximada.

El mejor resultado alcanzado, es decir, el valor más cercano al objetivo no se guarda en cada estado, sino como una variable global que se actualiza durante la búsqueda. Cuando se encuentra exactamente el objetivo, la exploración puede detenerse al haberse alcanzado la solución óptima. En cambio, si no existe solución exacta, se

conserva la mejor aproximación obtenida hasta ese momento y se continúa la exploración para comprobar si es posible mejorarla.

3.2.2 Generación de sucesores

En *La cifra exacta*, un estado puede generar varios estados sucesores al combinar dos cifras distintas del conjunto disponible mediante una de las operaciones permitidas. Para ello, se recorren las cifras activas y, para cada pareja posible, se prueban todas las operaciones aritméticas válidas en el concurso. Estos estados van generándose como paso previo a su exploración de manera explícita.

Gracias a que las cifras disponibles se mantienen siempre agrupadas en las primeras posiciones del array, es suficiente recorrer el conjunto una única vez con un doble bucle anidado. En cada iteración, se toma un par de posiciones y se intenta aplicar la operación en un orden, por ejemplo, elemento *i* con elemento *j*. Si no es posible (por dar un resultado no válido según las reglas), se invierte el orden y se vuelve a comprobar. De este modo, se consideran todas las combinaciones posibles de cifras y operaciones sin necesidad de generar permutaciones adicionales ni recorrer la misma pareja en ambos órdenes por separado.

En este proceso, las sumas y multiplicaciones solo requieren comprobarse una vez, ya que son operaciones conmutativas: verificar $i + j$ o $i \times j$ produce siempre el mismo resultado, por lo que evaluarlas en ambos órdenes generaría estados redundantes. En cambio, las restas y divisiones no son conmutativas; solo aportan un resultado válido en un sentido (división exacta y diferencia mayor que cero). Por ello, si no es posible realizar la operación de i sobre j, se prueba la inversa, de j sobre i, garantizando así que no se pierdan combinaciones válidas, pero evitando generar duplicados innecesarios.

Cada vez que una operación produce un resultado válido, se crea un nuevo estado en el que las dos cifras utilizadas se sustituyen por dicho resultado y la última cifra disponible del array, manteniendo las cifras agrupadas en las primeras posiciones, como se explicó anteriormente. Si el nuevo resultado mejora la mejor solución encontrada, se actualiza

el registro; y si coincide exactamente con el número objetivo, la búsqueda se detiene de inmediato.

La validez de un resultado depende tanto de las reglas aritméticas como de las restricciones del concurso: se descartan divisiones no exactas y resultados negativos.

3.2.3 Estrategias de búsqueda aplicadas

El árbol de estados se recorre utilizando dos estrategias principales: búsqueda en anchura y búsqueda en profundidad. Ambas se apoyan en la misma representación de estados y en el mismo mecanismo de generación de hijos, pero difieren en el orden en que se exploran.

La búsqueda en anchura avanza nivel por nivel, asegurando que la primera solución exacta encontrada utilice el menor número posible de operaciones. El proceso comienza insertando el estado inicial en una cola FIFO y estableciendo un contador que indica cuántos estados quedan por procesar en el nivel actual. Mientras la cola no esté vacía y no se haya encontrado una coincidencia exacta con el objetivo, se extrae el primer estado, se generan sus hijos y se añade cada hijo válido al final de la cola. Un nivel finaliza cuando el contador de hijos llega a cero; en ese momento, el contador de estados pendientes se actualiza al número de elementos de la cola y se incrementa el número de nivel. Este sistema garantiza que, si existe una solución exacta, se obtendrá con el mínimo número de pasos.

La generación de hijos se realiza recorriendo únicamente las cifras que siguen activas en el estado actual. Para cada pareja se intentan todas las operaciones permitidas. Las sumas y multiplicaciones solo se calculan en un orden, ya que su resultado no varía al intercambiar los operandos. En cambio, las restas y divisiones se prueban en ambos sentidos, pero únicamente cuando el segundo intento es necesario para obtener un resultado válido. Si la operación es posible, se crea un nuevo estado colocando el resultado en una de las posiciones utilizadas y trasladando la última cifra activa del array a la otra posición, manteniendo así todas las cifras disponibles juntas al inicio. Además,

se registran los operandos y la operación empleada, lo que permite reconstruir la secuencia completa al final.

En la búsqueda en profundidad, la exploración se realiza de forma recursiva, descendiendo por una rama hasta alcanzar un estado final antes de retroceder. En cada llamada, se recorren las cifras activas del array de forma anidada, seleccionando parejas y probando las operaciones con el mismo criterio de validez que en la búsqueda en anchura. Cuando una operación es posible, se calcula el resultado, se registran los operandos y la operación, y se reorganiza el array de la misma manera que en el caso anterior para mantenerlo compacto. A continuación, se llama recursivamente a la función para explorar el siguiente nivel.

El retroceso es un aspecto fundamental en la búsqueda en profundidad. Al volver de una llamada recursiva, se restauran las cifras a su estado anterior utilizando los valores guardados antes de modificar el array. Esto garantiza que el nivel actual permanezca intacto para continuar con el resto de las combinaciones posibles. No es necesario deshacer los registros de operaciones y operandos, ya que estos se sobrescriben en cada nivel. Igual que en la búsqueda en anchura, cada vez que se obtiene un resultado más cercano al objetivo, este se guarda; y si en algún momento se alcanza el número exacto, la exploración finaliza inmediatamente.

En la búsqueda en anchura, todos los estados generados en un nivel deben permanecer en memoria hasta que se haya terminado de procesar completamente dicho nivel, ya que el algoritmo avanza de forma ordenada nivel por nivel. Esto implica que, en instantes intermedios, la estructura de almacenamiento puede contener un número elevado de estados, lo que incrementa el consumo de memoria.

En cambio, la búsqueda en profundidad solo necesita mantener un único estado activo en cada momento. A medida que se avanza hacia niveles más profundos, se aplican operaciones sobre este estado para transformarlo en uno temporal que permita continuar la exploración. Al retroceder, dichas transformaciones se deshacen para restaurar el estado anterior y seguir explorando otras posibilidades. Este enfoque reduce drásticamente la cantidad de memoria necesaria como veremos en el capítulo 4,

aunque el orden de exploración no garantiza encontrar la solución en el menor número de pasos posibles.

Una variación sencilla de la búsqueda en anchura consiste en sustituir la cola FIFO por una cola de prioridad. En este caso, los estados se extraen no por orden de inserción, sino en función de un criterio definido, como la cercanía de la solución parcial al número objetivo. Este cambio transforma el recorrido en un algoritmo de ramificación y poda básico, ya que las ramas menos prometedoras quedan relegadas en la cola y, en la práctica, pueden no llegar a explorarse si se encuentra antes una solución óptima. Esta variante no se ha implementado en el trabajo.

3.3 Implementación para La palabra más larga

3.3.1 Generación del trie a partir del diccionario

La construcción del trie comienza con la lectura secuencial del archivo de diccionario, que contiene todas las palabras válidas para el juego. Cada palabra se procesa de forma individual: primero se convierte a minúsculas para unificar el tratamiento de las letras y garantizar que el mapeo posterior sea consistente. A continuación, se recorre la secuencia de caracteres desde la raíz del trie, avanzando por los nodos que ya existen cuando el prefijo correspondiente ha sido insertado previamente.

Cuando el camino actual no dispone de un nodo hijo para el siguiente caracter de la palabra, se crea uno nuevo y se enlaza al nodo padre en la posición correspondiente del array de hijos. Este array es de tamaño fijo y está indexado según el mapeo ASCII especial que sitúa la letra ‘a’ en el índice 0, mantiene contiguas el resto de letras y coloca la ‘ñ’ en la última posición. Cada posición de este array almacena un puntero, es decir, una dirección de memoria hacia el nodo hijo correspondiente. Si la posición está vacía (puntero nulo), significa que todavía no existe un hijo para ese caracter y debe crearse en ese momento.

Este diseño evita recurrir a estructuras asociativas más pesadas, como mapas o diccionarios, y permite que la inserción de cada caracter sea muy rápida. El coste

depende del tamaño del alfabeto, pero dado que en este caso se trata de un conjunto reducido y fijo de letras, el rendimiento en la práctica es prácticamente constante.

Durante la inserción, cada nodo del trie dispone de un campo booleano que indica si ese nodo corresponde al final de una palabra válida del diccionario. Cuando se completa la inserción de una palabra, el nodo alcanzado se marca con este indicador en true, de modo que la búsqueda pueda reconocer que el prefijo recorrido hasta ese punto constituye una palabra completa. Una vez insertadas todas las palabras, se recorre el trie de manera global para calcular dos propiedades esenciales para la exploración: la altura de cada nodo, definida como la longitud del camino más largo hasta una hoja descendiente; y el número de palabras alcanzables, que contabiliza cuántas palabras completas pueden formarse desde ese nodo.

Este recorrido posterior también sirve para construir, en cada nodo, distintas listas separadas, con sus hijos ordenados bajo diferentes criterios: orden alfabético, altura descendente, número de palabras alcanzables o heurísticas probabilísticas predefinidas. Gracias a este preprocesamiento, durante la resolución de un caso concreto no es necesario recalcular estas métricas ni reorganizar los hijos, lo que reduce el tiempo de exploración y permite seleccionar la estrategia de recorrido deseada.

3.3.2 Representación de estados y estructura de trie

En la palabra más larga, cada estado queda definido por dos estructuras: el prefijo de letras construido hasta el momento y el conjunto de letras todavía disponibles. El prefijo no es arbitrario: siempre corresponde a un camino válido dentro del diccionario. El estado mantiene una referencia al nodo del trie que representa ese prefijo; a la vez, el conjunto de letras disponibles se registra como un contador por letra que permite comprobar, en tiempo constante, si queda una unidad de una letra determinada antes de avanzar.

Para cubrir particularidades del alfabeto español, como la ñ, se emplea un mapeo directo basado en el código ASCII: a cada caracter, previamente convertido a minúscula, se le resta el valor ASCII de la letra ‘a’. De esta manera, las letras se distribuyen

en posiciones contiguas, alojando la 'a' en la posición 0 y el resto de los caracteres en posiciones consecutivas. La única excepción es la ñ, que se ubica de forma explícita en la última posición del array. Este esquema permite mantener un acceso directo y uniforme a cualquier hijo.

Durante la exploración de un caso, el estado avanza únicamente si se cumplen dos condiciones a la vez: desde el nodo actual existe en el trie una arista etiquetada con la letra candidata, y en el multiconjunto todavía queda al menos una unidad de esa letra. Cuando ambas se verifican, el prefijo se extiende hacia el hijo correspondiente y el contador de letras disponibles se decrementa en una unidad para esa letra. Este mecanismo asegura que la búsqueda nunca salga del diccionario y, al mismo tiempo, respete las restricciones del concurso (no usar más veces una letra de las que aparecen). La altura máxima efectiva del recorrido queda acotada por el número de letras inicialmente disponibles: con diez letras, la profundidad no puede superar diez, aunque el diccionario contenga palabras más largas.

3.3.3 Estrategia de exploración implementada

La resolución de la palabra más larga se apoya en una única estrategia de recorrido: una búsqueda recursiva en profundidad sobre el trie, guiada por distintos órdenes de exploración de los hijos y distintas podas. En este caso, las soluciones de interés se encuentran normalmente en niveles profundos del árbol por la propia naturaleza de la prueba. Por este motivo, una búsqueda en anchura resultaría ineficiente, pues obligaría a procesar exhaustivamente todos los niveles superiores manteniéndolos en memoria en orden riguroso. En cambio, la búsqueda en profundidad permite descender directamente por las ramas, evaluando de forma más temprana si llevan a palabras largas y reduciendo considerablemente el coste en espacio.

En cada paso, el algoritmo intenta extender el prefijo actual con una letra hija del nodo del trie, siempre que esa letra esté disponible en el multiconjunto del caso. Si la extensión es posible, se consume una unidad de esa letra, se avanza al hijo y se continúa recursivamente. Al volver, se restaura el contador, de modo que el estado del nivel anterior queda intacto para seguir probando alternativas.

El procedimiento arranca en la raíz del trie con el prefijo vacío y un contador de letras iniciales. En cada nodo, el algoritmo recorre la colección de punteros a hijos según el orden de la heurística seleccionada y, para cada hijo, aplica dos comprobaciones antes de descender. En primer lugar, verifica que queda al menos una unidad de la letra del hijo en el contador. En segundo lugar, aplica una poda por cota superior basada en la estructura del trie: si la longitud alcanzada en ese punto más la altura del nodo hijo no supera la longitud de la mejor solución encontrada, la rama se descarta sin explorarla. Esta cota se apoya en la métrica altura del nodo, que representa la máxima longitud adicional que podría alcanzarse continuando por su subárbol.

La implementación registra la solución parcial y actualiza la mejor solución cada vez que el nodo visitado marca fin de palabra y la longitud del prefijo supera la máxima registrada. El recorrido se interrumpe de inmediato cuando se alcanza una palabra de longitud diez, ya que el juego no permite construir palabras más largas con el conjunto de letras sorteadas.

El orden en que se exploran los hijos no es fijo; se selecciona a partir de diferentes listas ordenadas, generadas previamente a partir del mismo array de punteros durante la fase de preparación del trie. Cada lista aplica un criterio distinto, como priorizar nodos con mayor altura máxima (capaces de formar al menos una palabra más larga), seguir el orden alfabético, dar preferencia a nodos con mayor número de palabras alcanzables o utilizar un orden basado en probabilidades calculadas previamente.

En conjunto, la estrategia combina tres ideas fundamentales: limitar la exploración únicamente a aristas existentes en el trie, respetar en todo momento las restricciones del multiconjunto de letras disponibles y aplicar una poda por cota superior, que descarta aquellas ramas que no pueden superar la longitud de la mejor palabra encontrada hasta el momento.

Aunque la búsqueda subyacente es siempre recursiva en profundidad, el orden en el que se examinan los hijos de cada nodo puede variar, dando lugar a comportamientos muy distintos.

En primer lugar, se considera el recorrido en orden alfabético, que constituye la opción más simple y sirve como referencia base. En esta estrategia, los hijos de cada nodo se recorren según el orden natural de las letras en el alfabeto, sin aplicar ninguna priorización adicional.

Una segunda estrategia es la que prioriza los nodos con mayor número de palabras alcanzables. Para aplicarla, en cada nodo se calcula previamente cuántos descendientes terminales contiene, es decir, cuántas palabras completas pueden formarse a partir de él. Durante la exploración, se recorren primero los hijos con mayor valor en este contador, lo que favorece ramas con más potencial léxico.

Otra alternativa consiste en dar preferencia a los nodos con mayor altura máxima. La altura se define como la distancia en nodos hasta la hoja más profunda alcanzable desde el nodo actual. De este modo, los hijos con mayor altura representan caminos con más margen para crecer en longitud, lo que resulta útil cuando interesa encontrar palabras lo más largas posible.

Por último, se ha implementado una estrategia de prioridad probabilística, que se apoya en estadísticas obtenidas del propio diccionario. En la fase de carga se calculan dos matrices de probabilidades: una que estima la frecuencia con la que cada letra sucede a otra, y otra que mide la probabilidad de aparición de cada caracter en función de la profundidad en el árbol. Combinando ambas, con distintos pesos, se construye un criterio que ordena los hijos de cada nodo para favorecer secuencias que son más plausibles en la lengua.

Para priorizar la exploración de las ramas del trie, se emplean dos matrices de probabilidades precalculadas:

- Una matriz *P* de tamaño 26 por 26, en la que cada posición *(i,j)* almacena la probabilidad de que a la letra *i* le siga la letra *j*.
- Una matriz *N* de tamaño 10 por 26, en la que cada posición *(i,j)* almacena la probabilidad de que aparezca la letra *j* en el nivel *del* árbol (contando la raíz en el nivel 0).

Estas matrices se calculan una sola vez al leer el diccionario para añadirlo al trie y se guardan en un fichero de texto para futuras ejecuciones y estudio de probabilidades.

Antes de proceder a resolver los casos deseados, se realiza un recorrido recursivo de todos los nodos del árbol ordenando los vectores de hijos con un criterio basado en las probabilidades de ambas matrices.

Esta estrategia puede regularse ajustando los pesos que se le da a cada métrica, en este caso se otorga un 75% de importancia a la selección en base a la letra precedida y un 25% en base al nivel. Es importante especificar que hay partes del árbol donde solo puede usarse la probabilidad de niveles. Es el caso de los hijos de la raíz ya que no tiene un caracter previo.

Conviene señalar que este planteamiento no constituye una heurística especialmente sólida. El número de palabras accesibles desde cada hijo del trie es un criterio mucho más informativo para guiar la búsqueda, mientras que la estrategia probabilística resulta menos eficaz, ya que ignora que ya se ha fijado un prefijo de la palabra. El trie, en cambio, sí aprovecha esa información más restrictiva que la mera elección de la letra anterior. Se ha mantenido en la implementación de forma deliberada con el objetivo de contrastar sus resultados frente a estrategias basadas en información más detallada.

Aprovechando el cálculo de las probabilidades de que una letra preceda a otra parece interesante sacar un mapa de calor para poder ver gráficamente la estructura del diccionario:

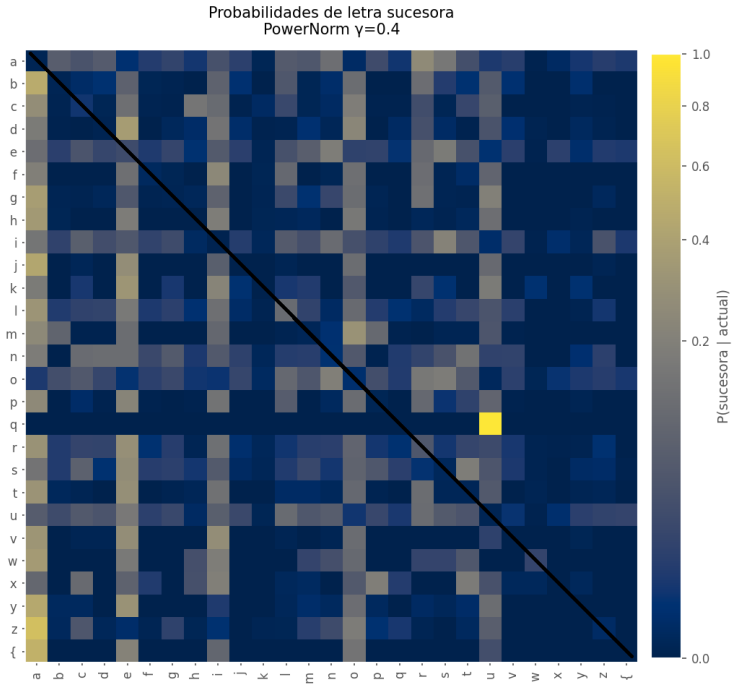

Figura 4. Mapa de calor de diccionario

En este mapa de calor, construido a partir de las palabras que consideramos soluciones válidas y almacenamos en nuestro trie, podemos radiografiar las reglas más características del vocabulario español y descubrir algunas menos evidentes para dirigir la heurística de exploración.

- La pareja q-u**.** Como es sabido la letra *q* siempre viene seguida de una letra *u*, reflejado por el color más pronunciado del mapa.
- Letras dobladas. Al tratarse de una matriz cuadrada, la matriz principal, marcada, muestra con que probabilidad la letra viene seguida de sí misma. Destacan dígrafos como «ll» y «rr». Algo menos intuitivo es la repetición de la letra '*e*', producido principalmente por los tiempos verbales con «ee ». en la raíz como *leer, creer o poseer* y por la combinación entre palabras que empiezan con '*e*' y un prefijo que acabe en *e* como *reescribir*. Además de otros dígrafos como «ch».
- Predominio de vocales. Las columnas que representan vocales son mas destacadas.

- Patrones de la lengua. Podemos observar la clara sucesión *m -> b, m -> p* y la no sucesión *n -> p , n -> b.* Además de ver como la letra z y *ñ* prácticamente solo se relacionan con vocales.

Un efecto llamativo es la probabilidad relativamente alta de que la 'w' se suceda a sí misma. Esto ocurre porque en nuestro vocabulario solo hay 80 palabras con 'w', y casos como *showwoman* generan porcentajes superiores al 1%. Aunque anecdótico, carece de relevancia práctica.

La idea es que nuestro algoritmo priorice la exploración de hijos a través de estas reglas sin necesidad de comprenderlas todas, basándose en las palabras del diccionario.

Capítulo 4 - Evaluación de los algoritmos

En este capítulo se presentan los resultados de las pruebas realizadas con las distintas implementaciones. El objetivo es comparar de forma clara cómo se comportan los algoritmos en los dos problemas estudiados: la prueba de cifras y la prueba de letras.

Para la comparación se han definido varias métricas: número de nodos generados, número de nodos visitados, profundidad alcanzada, tiempo de ejecución y memoria utilizada. Estas medidas permiten cuantificar el coste de cada ejecución del algoritmo y facilitan la comparación directa.

Además de los valores medios, se han analizado también las diferencias entre algoritmos en un mismo caso. De este modo se puede comprobar cuántas veces uno ofrece la solución antes que el otro, cuándo alcanzan la misma respuesta y en qué situaciones se observan ventajas claras.

Algunos resultados confirman lo que ya se podía deducir a partir de la propia definición de los algoritmos. Por ejemplo, que BFS encuentra siempre las soluciones en niveles más bajos, mientras que DFS tiende a explorar más en profundidad. Otros resultados son menos evidentes y solo se revelan al medirlos, como la profundidad típica en la que aparecen las mejores soluciones, la frecuencia con la que se alcanza exactamente el objetivo en la prueba de cifras o la eficacia relativa de cada heurística en la búsqueda de palabras dentro del trie.

El capítulo se organiza en dos apartados. En el primero se presentan los resultados del problema de cifras, centrados en la comparación entre BFS y DFS. En el segundo se estudia la prueba de letras, comparando las diferentes heurísticas de exploración aplicadas al trie.

4.1 Resolución de la prueba de Cifras

4.1.1 Preparación de los casos de prueba

Para poder comparar de forma justa el comportamiento de BFS y DFS fue necesario definir un conjunto de casos de prueba comunes. Todos los experimentos se han realizado sobre las mismas entradas, de manera que cada algoritmo resuelve exactamente los mismos problemas y las comparaciones no quedan condicionadas por diferencias en los datos.

Los casos de prueba se almacenan en archivos de texto, donde cada línea contiene seis cifras tomadas del conjunto válido del concurso (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100) y un número objetivo entre 100 y 999. Estos archivos son posteriormente cargados por BFS y DFS, garantizando que ambos algoritmos se ejecutan sobre idénticas condiciones de entrada.

La generación de los casos se lleva a cabo mediante un programa auxiliar desarrollado en Python. Este generador selecciona números al azar respetando las restricciones oficiales y vuelca cada instancia a un archivo .txt. De este modo, se pueden crear fácilmente grandes volúmenes de casos aleatorios.

Para el estudio experimental se han preparado un lote de 100.000 casos, lo que permite estudiar tanto su comportamiento medio como las diferencias en tiempo de ejecución, nodos generados y profundidad alcanzada.

4.1.2 Métricas utilizadas

Para evaluar el comportamiento de los algoritmos se han definido varias métricas orientadas a medir tanto la eficiencia temporal como la cantidad de nodos explorados en el árbol de estados.

Las diferentes métricas son:

- Número de vértices generados: cuenta cuántos estados hijos se crean a partir de las expansiones. Un mayor número indica más trabajo realizado en términos de construcción del espacio de búsqueda.

- Número de vértices visitados: contabiliza los estados que efectivamente son procesados, es decir, cuyos sucesores se han recorrido.
- Nivel máximo alcanzado: promedio de la mayor profundidad alcanzada por la búsqueda en los distintos casos de prueba.
- Nivel promedio de las soluciones: promedio de la profundidad a la que se encuentran las mejores soluciones en cada caso, lo que indica si suelen aparecer cerca de la raíz o en niveles más profundos.
- Tiempo de ejecución: duración total de la resolución de un lote de casos. Esta métrica permite comparar la eficiencia práctica de BFS y DFS más allá del número de nodos. Para interpretar correctamente estos valores es necesario especificar el entorno de pruebas empleado. En nuestro caso, todas las ejecuciones se han realizado en un equipo con las siguientes especificaciones de hardware y software: procesador AMD Ryzen 9 7900X de 12 núcleos y 24 hilos a 4.7 GHz, 64 GB de memoria RAM y sistema operativo Windows 11 Pro (compilación 26100).
- Memoria utilizada: se estima a partir del número de estados que cada algoritmo necesita mantener simultáneamente en memoria. BFS almacena todos los nodos pendientes de explorar en una cola, mientras que DFS mantiene únicamente un estado activo y la pila de recursión, lo que genera diferencias notables.

4.1.3 Comparación BFS y DFS

Una vez definidas las métricas, se procedió a comparar directamente el comportamiento de BFS y DFS sobre el lote de 100.000 casos. Dado que ambos algoritmos exploran el mismo espacio de estados, resuelven necesariamente las mismas entradas; por tanto, las diferencias observadas se deben únicamente a la estrategia de búsqueda empleada.

Los primeros resultados muestran diferencias claras en el número de nodos generados y visitados. DFS, al recorrer una rama completa antes de retroceder, tiende a generar menos vértices en total. En los experimentos, esta reducción se sitúa en torno a un 30%. Además, DFS visita prácticamente todos los nodos que genera, ya que cada nuevo

estado se procesa de inmediato mediante recursión y solo se crean hijos en el momento exacto en que van a ser explorados. En contraste, BFS produce un número mayor de estados, muchos de los cuales nunca llegan a visitarse. Esto ocurre porque, si aparece una solución en un nivel dado, los demás vértices ya generados en ese mismo nivel quedan en la cola sin ser procesados, lo cual es el comportamiento esperado de este algoritmo.

Algoritmo DFS		Algoritmo BFS		
Promedio vértices generados	Promedio vértices visitados	Promedio vértices generados	Promedio vértices visitados	Número de casos
226.460,12	226.460,12	296.883,87	180.679,45	100.000

Tabla 1. Estadísticas de vértices.

El análisis de profundidades refuerza esta diferencia. DFS alcanza casi siempre el nivel máximo posible (cinco operaciones, correspondientes a la reducción progresiva de seis cifras iniciales hasta una única cifra final). BFS, en cambio, suele detenerse en niveles más cercanos a la raíz, normalmente entre dos y tres operaciones. Esto ocurre porque su naturaleza garantiza que la primera solución encontrada es siempre la más corta en número de pasos. Esta característica tiene una consecuencia práctica inmediata: BFS es especialmente útil cuando interesa obtener la solución que utiliza el menor número de operaciones posibles. DFS, en cambio, explora en profundidad y tiende a encontrar rápidamente soluciones parciales muy cercanas al número objetivo, aunque requieran más pasos.

Algoritmo DFS		Algoritmo BFS		
Promedio niveles máximos alcanzados	Nivel promedio de las soluciones	Promedio niveles máximos alcanzados	Nivel promedio de las soluciones	Número de casos
4,99	3,79	2,98	2,82	100.000

Tabla 2. Promedio de niveles.

También se ha realizado una comparación acerca del comportamiento de BFS y DFS en los mismos casos individuales. Esta comparación permite comprobar, por un lado, cuál de los dos algoritmos resuelve antes en tiempo (es decir, cuál completa primero la exploración hasta dar con una solución) y, por otro, cuál obtiene una expresión que requiere un menor número de operaciones aritméticas. En un lote de 1000 casos, DFS fue más rápido en 666 ocasiones, mientras que BFS generó una solución estrictamente más corta en 650. En 350 casos ambos algoritmos alcanzaron expresiones de la misma longitud. Estos resultados reflejan la propia naturaleza de las dos estrategias: BFS prioriza obtener soluciones con menos operaciones, mientras que DFS se caracteriza por resolver los casos con mayor rapidez.

	Menor para BFS	Empate	Menor para DFS
Número de operaciones	650	350	0
Tiempo	0	334	666

Tabla 3. Métricas compilatorias DFS y BFS

En cuanto al tiempo de ejecución, las diferencias son aún más marcadas. DFS resuelve los lotes con tiempos muy inferiores, llegando a ser hasta siete veces más rápido que BFS resolviendo 100.000 problemas. La explicación se encuentra en la gestión de estados. DFS trabaja sobre un único estado que se transforma recursivamente, aplicando y deshaciendo operaciones sobre la misma estructura, lo que evita copias innecesarias. BFS, en cambio, necesita mantener en memoria todos los estados generados en un nivel

hasta que llega el momento de explorarlos. Esto obliga a copiar estructuras completas y almacenar simultáneamente un gran número de vértices en la cola, con el coste en tiempo que ello conlleva.

Tiempo algoritmo DFS	Tiempo algoritmo BFS	
5 min 21 s 225 ms	35 min 13 s 937 ms	100.000 casos
3,21225 ms	21,13937 ms	promedio en 1 caso

Tabla 4. Tiempos de resolución de casos.

Este comportamiento repercute directamente en el consumo de memoria. Las estimaciones muestran que BFS puede llegar a usar cientos de veces más espacio que DFS en los lotes grandes. La razón es sencilla: DFS solo necesita la pila de llamadas y un estado activo, mientras que BFS acumula en memoria todos los vértices de cada nivel, incluso si nunca se van a recorrer.

En resumen, DFS ofrece ventajas claras en tiempo y memoria, mientras que BFS, aunque requiere un mayor consumo de memoria, resuelve siempre con el menor número de operaciones y en un tiempo muy inferior al que dispone un concursante humano. Las mediciones realizadas con las herramientas de *Visual Studio* muestran que el pico de uso de memoria en la ejecución del lote de 100.000 casos alcanza los 254 MB (figura 5), frente al consumo mucho menor registrado por DFS (figura 4). Aun así, esta cantidad resulta perfectamente asumible en un equipo moderno. Conviene señalar que estas cifras no reflejan únicamente el coste intrínseco del algoritmo, sino también la sobrecarga del propio entorno de ejecución (estructuras auxiliares, gestión de la pila, bibliotecas, etc.). Aun así, resultan útiles para comparar el orden de magnitud de los costes espaciales.

Figura 5. Uso de memoria en DFS

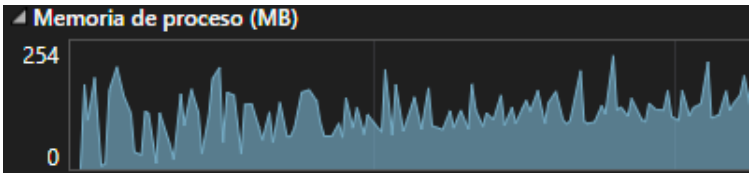

Figura 6. Uso de memoria en BFS.

El análisis de costes resulta esencial para comprender las limitaciones y el comportamiento real de los algoritmos implementados. Dicho análisis se basa en dos componentes: el número de estados posibles que se pueden generar de acuerdo con las reglas del concurso y el coste de generar y procesar cada estado. En consecuencia, el coste total puede expresarse como:

$$T(n) = \#estados \times coste\ por\ estado$$

donde n representa el número de cifras iniciales (en este caso, 6).

El coste por estado no se limita únicamente a aplicar una operación sobre dos cifras, sino que incluye también la generación de todos los hijos posibles. Para ello, el algoritmo recorre de manera sistemática todas las parejas de cifras activas mediante un doble bucle, y sobre cada pareja prueba hasta cuatro operaciones aritméticas. Este procedimiento implica un trabajo proporcional al número de combinaciones de dos cifras, es decir, a $\binom{n}{2}$. Una vez elegida la pareja y la operación, tanto el cálculo del resultado como la reorganización del array de cifras se realizan en tiempo constante. En términos generales, por tanto, generar todos los hijos de un estado supone un coste proporcional a $\binom{n}{2}$, aunque en la práctica, dado que el número de cifras iniciales está acotado a seis, este valor nunca supera las quince combinaciones posibles y puede considerarse prácticamente constante. De este modo, el coste total en tiempo sigue siendo proporcional al número de estados generados:

$$T(n) = O(\#estados)$$

La diferencia fundamental entre ambas estrategias aparece en el coste en espacio. Mientras que el tiempo depende directamente del número total de estados generados, el espacio depende del número de estados que deben mantenerse simultáneamente en memoria y del tamaño de cada uno.

Cada estado se representa con la estructura `Problema`, que contiene la solución parcial alcanzada, el registro del orden de uso de cifras, las operaciones realizadas y el conjunto de candidatos disponibles:

```
constexpr short CIFRAS_INICIALES = 6;
constexpr short CIFRAS_MAXIMAS_ENCADENADAS = 2 * CIFRAS_INICIALES;

struct Problema {

	//[...]

	int _solParcial;
	std::array<int, CIFRAS_MAXIMAS_ENCADENADAS> _ordenDeUso;
	std::array<char, CIFRAS_INICIALES> _operacionesEnOrden;
	std::array<int, CIFRAS_INICIALES> _numCandidatos;
};
```

Cada componente ocupa:

- _solParcial: un valor numérico intermedio, 4 bytes.
- _ordenDeUso: array de 11 × 4 = 44 bytes.
- _operacionesEnOrden: array de 6 × 1 = 6 bytes.
- _numCandidatos: array de 6 × 4 = 24 bytes.

El tamaño teórico en memoria de cada estado para los algoritmos de cifras es: 4 + 44 + 6 + 24 = 78 bytes. Este valor corresponde a la suma directa de los campos, pero no tiene en cuenta el alineamiento de memoria [22] que introduce el compilador. Al medirlo directamente en C++ mediante sizeof (Problema), el tamaño real obtenido fue:

```
sizeof(Problema)=84 bytes
```

Con este valor, el coste espacial puede expresarse como:

$$M(n) = \#\text{estados simultáneos} \times \text{tamaño de un estado}$$

En DFS, únicamente se mantiene en memoria el estado actual junto con la pila de recursión. Como la profundidad máxima del árbol de cifras es 5, la cantidad de estados vivos simultáneamente está muy acotada y el consumo de memoria resulta despreciable.

En cambio, en BFS la situación es muy distinta. Este algoritmo necesita mantener en la cola todos los estados generados en un mismo nivel hasta que hayan sido procesados. Esto implica que, aunque el número total de estados generados a lo largo de la búsqueda sea similar en BFS y DFS, la cantidad de estados coexistiendo en memoria en un momento dado es mucho mayor en BFS.

En el peor caso con 6 cifras, la cola llega a acumular hasta 2.764.800 estados simultáneos (ancho máximo en el último nivel, como se calculó en el Capítulo 3). El consumo de memoria asciende a:

$$2.764.800 \times 84\ bytes \approx 226\ MB$$

En DFS, por el contrario, solo se mantiene un camino activo en la pila de recursión. La profundidad máxima del árbol de búsqueda con *n* cifras iniciales es $n - 1$, ya que en cada operación se reduce en una la cantidad de cifras activas. Esto significa que, como mucho, se almacenan simultáneamente $n - 1$ estados (uno por cada nivel de profundidad). Dado que cada estado ocupa 84 bytes, el consumo máximo de memoria simultánea en DFS es:

$$M_{\mathrm{DFS}}(n) \leq (n-1) \times 84 \text{ bytes}$$

Para el caso concreto de $n = 6$ cifras iniciales, la profundidad máxima es 5 y el consumo teórico se limita a:

$$M_{\mathrm{DFS}}(6) \leq 5 \times 84 = 410 \text{ bytes}$$

Estos cálculos coinciden con las mediciones experimentales mostradas en las figuras 4 y 5, donde BFS alcanza un consumo cercano a los 254 MB y DFS se mantiene en el orden de kilobytes. Aunque el coste en tiempo y memoria de BFS resulta mucho mayor que el

de DFS, la magnitud absoluta sigue siendo perfectamente asumible para resolver la prueba dentro de los recursos disponibles en cualquier equipo actual.

4.2 Resolución de la Palabra más larga

4.2.1 Preparación de los casos de prueba

Para comparar de manera justa las diferentes estrategias de exploración en la prueba de letras fue necesario definir un conjunto de casos de prueba comunes. Todos los algoritmos se han ejecutado sobre las mismas entradas, garantizando que las diferencias observadas provienen únicamente del criterio de exploración aplicado en el trie.

Los casos de prueba se almacenan en archivos de texto que contienen, en cada línea, diez letras seleccionadas aleatoriamente bajo las restricciones del concurso: deben aparecer entre tres y seis vocales, mientras que el resto se completa con consonantes. Estas condiciones reproducen fielmente las reglas del programa y permiten que los experimentos sean representativos.

La generación de los casos se realiza mediante un programa auxiliar desarrollado en Python. Este programa selecciona letras de manera aleatoria respetando la proporción de vocales y consonantes.

Para los experimentos se han creado dos conjuntos de pruebas diferenciados. El primero contiene casos completamente aleatorios, en los que las letras pueden dar lugar a palabras de cualquier longitud. El segundo se centra exclusivamente en instancias cuya solución es una palabra de diez letras extraída directamente del diccionario. Estos últimos permiten analizar con mayor precisión el comportamiento de las distintas heurísticas cuando se alcanza la longitud máxima posible en el juego, ya que la búsqueda se detiene inmediatamente al encontrar una palabra de diez letras.

4.2.2 Comparación de heurísticas

Todas estas estrategias se aplican sobre el mismo conjunto de casos de prueba, lo que permite comparar directamente su rendimiento y evaluar hasta qué punto los distintos

órdenes de exploración modifican el número de nodos recorridos, la rapidez en alcanzar soluciones largas y el tiempo total de ejecución.

Estrategia de Exploración	Total nodos explorados	Promedio nodos completamente explorados	Tiempo (s)
Orden alfabético	262.596.408	262,59	59s
Palabras alcanzables	260.726.408	260,72	53s
Mayor altura	265.876.654	265,87	55s
Probabilidades	263.246.178	263,24	59s

Tabla 5. Estrategias en el diccionario completo

Como podemos observar ninguna estrategia es significativa ni en tiempo, ni en ahorro de nodos visitados, esto sucede porque la mayoría de las palabras de nuestro vocabulario son de longitud 6 y los algoritmos recorren la mayor parte del árbol por si hubiera una solución mejor.

Es por esto por lo que se decide realizar la prueba solo para las palabras de 10 letras de nuestro diccionario, ya que al encontrar una palabra de 10 letras y no poder mejorar la solución, se detendrá la búsqueda y podremos cotejar mejor los nodos visitados.

El diccionario completo contiene 647.162 palabras, de las cuales 99.195 corresponden exactamente a 10 letras. Estas últimas se extrajeron y desordenaron antes de resolver los casos.

Exploración de los 99.195 casos con 10 letras.

Estrategia de Exploración	Total nodos explorados	Promedio nodos completamente explorados
Orden Alfabético	27.933.117	273,51
Palabras alcanzables	26.946.680	263,00
Mayor altura	27.524.463	268,88
Probabilidades	29.971.477	294,03

Tabla 6. Estrategias en palabras de 10 letras

En la tabla 6 se compara el número medio de actualizaciones de la solución para cada estrategia de exploración en dos escenarios: conjuntos de letras con soluciones aleatorias y conjuntos desordenados cuya solución son palabras de longitud máxima (10 letras) para obtener una visión más detallada de cómo cada estrategia se aproxima a la solución.

Estrategia de Exploración	Casos con soluciones de 10 letras desordenadas	Casos aleatorios
Orden alfabético	7,37	4,76
Palabras Alcanzables	7,34	4,61
Mayor altura	7,19	4,64
Probabilidades	9	5

Tabla 7. Actualizaciones de la solución

4.3 Discusión conjunta

En la prueba de cifras se confirma que la elección del algoritmo es determinante. BFS siempre devuelve la solución con menos operaciones, pero lo hace a costa de un coste en tiempo y memoria muy alto. DFS, en cambio, es mucho más rápido y ligero, aunque suele generar expresiones más largas. La diferencia entre ambos no depende de los datos concretos, sino de su naturaleza: BFS explora nivel a nivel y acumula estados, mientras que DFS profundiza de inmediato y reutiliza la misma estructura de forma recursiva.

En la prueba de letras la situación es distinta. Aquí todas las variantes comparten la misma búsqueda en profundidad, y lo que cambia es únicamente el criterio de ordenación de los hijos en el trie. Por ello, las diferencias en tiempo y nodos visitados son mucho menores. Aun así, las heurísticas marcan diferencias en la forma de explorar: las basadas en altura o en número de palabras alcanzables favorecen caminos con mayor potencial de crecer en longitud, mientras que la heurística probabilística resulta menos eficaz, ya que prescinde de información estructural y emplea un criterio estrictamente menos informado.

Comparando ambas pruebas se obtiene una visión complementaria de cómo influye la estrategia de exploración en problemas de búsqueda. En cifras, donde el espacio es más reducido pero la calidad de la solución puede variar, la elección entre anchura y profundidad marca diferencias en tiempo y memoria. En letras, en cambio, el espacio de búsqueda es enorme pero las soluciones son relativamente homogéneas, y lo que resulta relevante es el criterio con el que se ordenan los caminos dentro de la misma estrategia de profundidad. En conjunto, los experimentos muestran que la eficiencia de la búsqueda no depende solo del tamaño del espacio de estados, sino de qué información se aprovecha para guiar la exploración en cada contexto.

Capítulo 5 - Interfaz web de la herramienta

Este capítulo tiene como propósito presentar la interfaz a través de la cual es posible ejecutar las funcionalidades desarrolladas en el proyecto desde el propio navegador. Para ello, se realizará una comparación entre el funcionamiento del programa televisivo y de la implementación del proyecto en los mismos escenarios.

Al acceder a la página web se observan dos rutas diferentes: una para la prueba de cifras y otra para la de letras. La estética general, especialmente en lo relativo a la paleta de colores, mantiene una gran similitud con la del programa televisivo.

En la prueba de *La cifra exacta* se presentan seis espacios en blanco destinados a los números con los que operar. El usuario dispone de dos posibilidades: generar tanto las cifras como el número objetivo de manera aleatoria mediante un botón, o bien introducirlos manualmente. En la primera opción, se sigue el mismo criterio que en el programa televisivo, empleando números del 1 al 10 (los denominados "números pequeños") junto con 25, 50, 75 y 100 (los llamados "números grandes"). El número objetivo, por su parte, se genera dentro del intervalo de 100 a 999.

Por otra parte, al introducir los valores manualmente, se amplían las posibilidades respecto al formato original del programa, ya que pueden incluirse cifras fuera de los rangos establecidos. Del mismo modo, el número objetivo puede superar los límites establecidos en el concurso televisivo.

Cambiando a la prueba de La palabra más larga, al igual que en la anterior existen dos posibilidades: generar las letras aleatoriamente o introducirlas manualmente. En el programa televisivo se aceptan tanto plurales como formas verbales. Además, la implementación añade un enlace a la definición de la palabra hallada en la versión web de la RAE.

5.1 La Interfaz

Al acceder a la herramienta, lo primero que se puede observar es una página principal en la que aparecen dos botones para cada una de las pruebas: uno en color amarillo para la prueba de cifras; y otro de color rojo para la prueba de letras.

Figura 7.Vista principal interfaz web.

Una vez seleccionada una de ellas, la interfaz redirige a cada prueba.

En la prueba de cifras, aparece una pantalla de color azul con siete huecos en blanco, seis en la parte superior y otro en la parte inferior. Tal y como se indica en la descripción («escribe las seis cifras con las que operar»), los seis huecos de la parte superior están destinados a las seis cifras con las que se operará para lograr el número objetivo, que se indica en el hueco en blanco de la zona inferior.

Estas cifras, como se ha mencionado previamente, pueden introducirse de forma manual o aleatoriamente al seleccionar la opción de «genera aleatorios». Para resolver es importante tener en cuenta que tras poner los números hay que seleccionar el botón de «Resolver». Asimismo, hay un botón específico para borrar las cifras en caso de querer introducir unas diferentes.

Como se puede observar, en la parte superior hay un panel desde el cual podemos acceder a la página de inicio al pulsar «home» o directamente a la prueba de letras al seleccionar dicha opción.

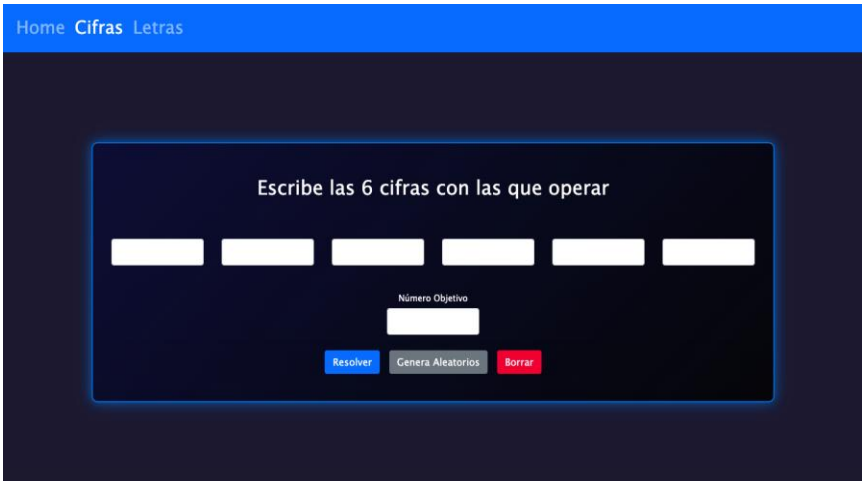

Figura 8.Vista Prueba Cifras Interfaz

Al acceder a la prueba de letras, aparece también una pantalla azul, en este caso con diez huecos en blanco. Al igual que en la prueba de cifras, aparece una descripción («escribe 10 letras») que indica que los diez huecos se destinan a escribir una letra en cada uno de ellos. En esta prueba, como se ha mencionado y de la misma manera que en la prueba de cifras, se pueden introducir las letras manualmente o generarse aleatoriamente al seleccionar «generar aleatorios». Para conseguir la palabra seleccionamos el botón de resolver y para borrar e introducir nuevos caracteres pulsamos la opción de «borrar».

Figura 9. Vista Prueba Letras Interfaz

5.2 Comparación con el programa

A continuación, se comparan las respuestas de la interfaz y del programa en ambas pruebas.

5.2.1 La palabra más larga

Figura 10. Caso 1. Solución Programa 1

Figura 11. Caso 1. Solución Programa 2

Figura 12. Solución caso 1 Interfaz web

En este caso, podemos comprobar cómo los participantes del concurso no alcanzan una palabra de longitud 9, a diferencia de la implementación, que encuentra una palabra más larga con las letras disponibles.

Figura 13. Caso 2. Solución programa

Figura 14. Caso 2. Solución caso 2. Interfaz web

En este otro caso, podemos comprobar que, a pesar de que los concursantes no logran una palabra con longitud de 7 letras (quedándose muy cerca con la palabra *puntúo*), la presentadora nos plantea como palabra más larga la misma que ofrece nuestra implementación.

5.2.2 La cifra exacta

Figura 15. Caso 1. Cifras y número objetivo

Figura 16. Caso 1. Solución Programa

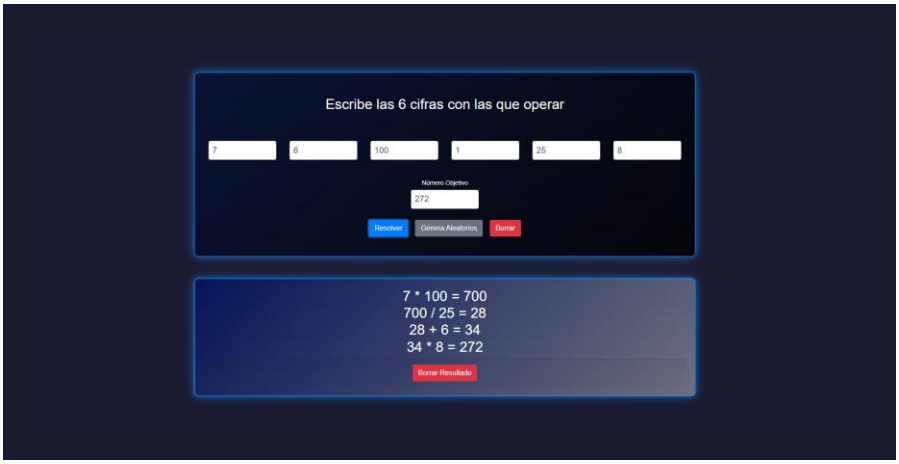

Figura 17. Caso 1. Solución interfaz web

En este caso, podemos comprobar como nuestra implementación consigue un resultado igual realizando menos operaciones de las que se realizan en el concurso.

Figura 18. Caso 2. Cifras y número objetivo

Figura 19. Caso 2. Solución Programa

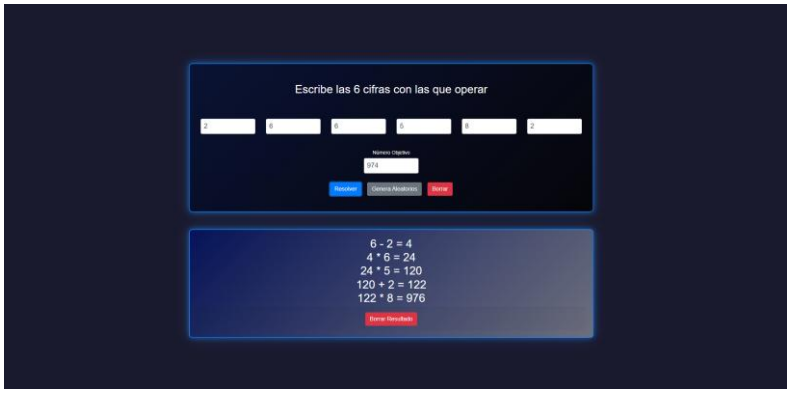

Figura 20. Caso 2. Solución Interfaz Web.

En este otro caso, no se puede alcanzar la cifra exacta, sin embargo, podemos conseguir una cifra aproximada. En el concurso, podemos ver que el concursante alcanza una aproximación al número 968. Nuestra implementación alcanza también una aproximación a dos números de diferencia, al alcanzar el número 976.

Capítulo 6 - Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado y se señalan posibles líneas de desarrollo futuro. Primero se resumen los principales resultados alcanzados, y a continuación se plantean mejoras y ampliaciones que podrían enriquecer la herramienta en etapas posteriores.

6.1 Conclusiones

En primer lugar, uno de los objetivos del proyecto era desarrollar una herramienta capaz de resolver de forma automatizada las pruebas de La cifra exacta y La palabra más larga del programa Cifras y Letras, aplicando algoritmos de búsqueda y estructuras de datos. Dicho objetivo se ha alcanzado de manera satisfactoria, puesto que se han logrado implementaciones funcionales que reproducen y amplían las dinámicas del concurso.

En relación con dicho objetivo, se ha analizado el comportamiento y el rendimiento de diferentes soluciones algorítmicas, comparando diversas alternativas e identificando sus ventajas y limitaciones en términos de tiempo de ejecución, exactitud y adaptabilidad.

Asimismo, el proyecto ha logrado otro de sus objetivos fundamentales: acercar estas implementaciones al usuario a través del desarrollo de una interfaz web. Para ello, se ha recurrido a la integración del código C++ con WebAssembly, lo que permite la ejecución directa de los algoritmos en el navegador sin necesidad de instalaciones adicionales ni configuraciones externas. Esta decisión garantiza la accesibilidad y facilidad de uso, factores determinantes para el éxito de nuestro proyecto.

En cuanto a la percepción del usuario, se ha procurado mantener una estética similar a la del programa televisivo, favoreciendo la familiarización y la inmersión en el entorno. No obstante, la implementación permite introducir cifras y condiciones más allá de las contempladas en el formato original, además de consultar automáticamente el significado de las palabras generadas en el diccionario de la RAE.

En el proyecto, la combinación de técnicas algorítmicas con una interfaz accesible posibilita la reproducción y realización de procesos tradicionalmente manuales. Asimismo, la elección de C++ como lenguaje base, junto con la compilación hacia WebAssembly, ha posibilitado integrar soluciones de alto rendimiento en entornos web, abriendo la puerta a futuras aplicaciones en contextos distintos al del concurso.

En definitiva, el proyecto ha alcanzado sus objetivos: ha desarrollado una herramienta funcional y accesible para resolver de forma automatizada las pruebas de Cifras y Letras, se ha estudiado el rendimiento de diversas alternativas y se ha desarrollado una interfaz web a la implementación capaz de trasladar las soluciones a un entorno interactivo.

6.2 Trabajo futuro

Como opciones de futuro surgidas a lo largo del desarrollo del proyecto, cabe hacer referencia a dos campos en los que se puede trabajar, uno dedicado a la implementación y al análisis de distintos algoritmos; y otro relacionado con la interfaz web.

En primer lugar, en relación con un futuro análisis de distintos algoritmos, se plantean diversas líneas de mejora. Entre ellas cabe destacar: la incorporación de nuevas modalidades de juego que incluyan las restantes pruebas del concurso, tanto en el ámbito matemático como lingüístico, susceptibles de abordarse mediante técnicas computacionales; la ampliación de la prueba de cifras para admitir números de mayor longitud, tanto en el número objetivo como en las cifras disponibles; y la implementación de la prueba de letras en otros idiomas, ampliando así el alcance de la aplicación y adaptándola a contextos lingüísticos diversos mediante el uso de diccionarios y normas específicas de cada lengua. Por otra parte, en lo relativo a la interfaz web, se identifican varios aspectos de interés para desarrollar en el futuro. Entre ellos se incluyen: crear una versión en la que se puedan seleccionar las operaciones disponibles, con fines didácticos, permitiendo además restringir los valores; implementar un modo de juego en línea que favorezca la interacción entre usuarios; desarrollar un sistema de ranking de jugadores que incentive la competitividad; y, finalmente, incorporar un espacio

colaborativo para la comparación de algoritmos, en el que distintos desarrolladores puedan contrastar sus propuestas de resolución y evaluar su rendimiento en un entorno común.

Introduction

Background and Game Rules

The game show *Le Mot le plus long* debuted on French television in 1965 as a vocabulary contest focused on finding the longest word from a limited set of letters. In 1972, a second challenge centered on mental arithmetic was added, and the program adopted its definitive name: *Des chiffres et des lettres* [1] (*Numbers and Letters*). Since then, it has been broadcast almost continuously in France for more than fifty years and it has been adapted in other countries, such as the United Kingdom (Countdown, since 1982), and Spain, where it premiered in 1991. In Spain, it has gone through several stages and it is currently back on TVE's programming schedule.

Throughout its history, the contest has maintained two fundamental challenges at its core: the longest word and the exact number, pillars that have remained practically unchanged since its inception. However, in certain stages and versions, the program has incorporated other complementary challenges, such as rapid mental arithmetic tasks, themed vocabulary challenges, or even linguistic minigames like hidden words or cross definitions, in order to enrich the format and adapt it to new audiences. These additional challenges usually occupied a smaller portion of the program and varied depending on the edition or the country in which it was broadcast.

This work focuses exclusively on the two classical and most representative challenges of the contest, addressing them from an algorithmic and computational perspective, with the goal of automating their solution and analyzing the most efficient strategies to tackle the challenges they pose from a software development point of view.

In The Exact Number challenge, six numbers are selected, randomly drawn from two groups: the so-called "small numbers" (values from 1 to 10, which may repeat) and the "large numbers" (25, 50, 75, 100).From these six initial values, the goal is to construct, using basic arithmetic operations (+, −, ×, ÷), an expression that results in a randomly chosen target number between 100 and 999.

Each number can be used as many times as it appears. Intermediate results can also be combined, but only operations that produce positive integer results are allowed.

The player has a limited time (between 30 and 45 seconds, depending on the version) to find a valid expression. If none of the contestants finds the exact solution, the closest solution to the target is accepted.

For example, if the available numbers are {3, 4, 5, 6, 6, 25} and the target number is 597, the best solution would be:

Operaciones:	Cifras disponibles:
25×4= 100	{3,5,6,6,100}
100×6=600	{3,5,6,600}
600-3=597	{5,6,597}

solución: ((25 × 4) × 6) − 3 = 597

In The Longest *Word* challenge, contestants choose at the beginning how many vowels they want to include in the set of ten drawn letters, with the condition that this number is between a minimum of three and a maximum of six. The remaining letters, up to ten, are taken from consonants. As in the original version, letters can be repeated but can only be used as many times as they appear in the draw. Using these letters, the player must form the longest possible word recognized by the dictionary of the language in which the game is played. In the case of Spanish, the dictionary of the Real Academia Española (RAE) [2] is used, and plurals and verb forms are also accepted.

For example, from the set of letters {U, Q, W, Q, H, U, R, M, O, E}, valid Spanish words would include *quorum*, *humero*, or *húmero*, regardless of the fact that the initial set does not include diacritical marks, which may appear in the final word.

Motivation

The project originates from the interest in addressing the classic *Cifras y Letras* [3] challenges (*Countdown* in its English version), which combine logical reasoning and language skills in an engaging format. These challenges have inspired numerous digital implementations over time, both as desktop software and web applications. Examples

include *CaLao* [4] and *Compteurs de mots* [5], which reproduce the contest rules and automatically solve its two main challenges.

The aim of the project is not to propose an unprecedented solution, but rather to adopt a rigorous Software Engineering perspective to design, compare, and refine different solving strategies. The focus is to analyze aspects such as algorithmic efficiency, system extensibility, and clarity in result representation. In *The Exact Number* challenge, the project emphasizes the application and comparison of search techniques such as BFS and DFS [6], with special attention to their impact on execution time and resource consumption. In *The Longest Word* challenge, the use of data structures such as tries [7] is explored, along with the incorporation of heuristics aimed at guiding the search towards more promising solutions.

In this way, the project brings together the technical dimension of algorithm design, implementation, and analysis with the cultural and playful appeal of a well-known TV show. Ultimately, it constitutes a formative exercise that enables the practical application of knowledge acquired during the degree, while also encouraging experimentation with different approaches to solving classic combinatorial challenges.

Goals

The main objective of this project is to address, from an algorithmic perspective, the two fundamental challenges of the *Cifras y Letras* contest. The overall goal is to develop computational solutions that can efficiently solve these challenges, exploring various techniques learned during the Software Engineering degree.

To begin with, the project aims to compare different solving strategies: *The Exact Number* challenge and *The Longest Word* challenge. In the case of *The Exact Number* challenge, the goal is to analyze and experiment with different approaches based on classical search algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS), as well as different pruning and traversal control techniques. The intention is to identify which approach provides the best results based on exploration times, depth reached, and nodes visited.

A parallel objective is the formal implementation of a solution for the letter challenge. This is done through a custom approach, relying on data structures such as prefix trees (tries) and efficient search techniques. This design aims to provide a solution capable of solving the problem in a reasonable amount of time despite the extensive Spanish vocabulary.

In addition to the algorithmic objectives, the project includes a browser-accessible interface. This web page allows users to execute both challenges easily, displaying the results obtained for each case. The main purpose is to allow any user, without technical knowledge, to test the system, compare solutions, and, if desired, contrast them with actual episodes of the television program. In this way, the tool not only makes the project more accessible but also allows its performance to be evaluated in a practical and recognizable context.

Finally, this work aspires to serve as an integrative exercise of the knowledge acquired during the degree, covering everything from code structuring to the application of data structures and classical problem-solving patterns. The project is conceived as an opportunity to apply this learning in a complete environment, with measurable results and real possibilities for improvement and expansion.

Work plan

The project was approached using an incremental methodology, in which each advancement was integrated into the overall system and evaluated as a whole. This strategy has allowed the early identification of errors, comparison of different approaches, and progressive adjustment of the system architecture based on the results obtained.

The main stages of development were:

- Phase 1: Problem study and formal definition of the rules. A detailed analysis of the two main contest challenges has been developed: *The Exact Number* and *The Longest Word*, establishing the technical requirements and specific constraints of each.

- Phase 2: Design of solution algorithms for the numbers challenge. Classical algorithms such as DFS and BFS were applied, analyzing their behavior and efficiency in problem solving.
- Phase 3: Construction of an efficient structure for the letters challenge. A structure based on prefix trees (tries) was designed and implemented, on which different traversals, pruning methods, and heuristics were applied to compare efficiency in word generation and validation.
- Phase 4: Development of a basic web interface. An accessible and functional interface was designed using HTML, CSS, Bootstrap [8], and JavaScript, allowing both challenges to be executed from the browser and results to be displayed clearly.
- Phase 5: Integration of the C++ engine with the web interface. Using Emscripten [9], the C++ code was compiled to WebAssembly [10], enabling the complete system to run directly in the user's browser.
- Phase 6: As a potential extension, the incorporation of a third challenge based on Natural Language Processing (NLP) was studied. The proposal consisted of using vector representations (semantic embeddings) to compare definitions. This approach was explored experimentally, but it was finally decided to discard it as it did not fully align with the objectives nor offer satisfactory results within the framework of the project.
- Phase 7: Final integration and testing. System-wide tests were carried out to verify stability, check the consistency of solutions, and validate the performance of the different algorithms in real situations.

During development, various environments and tools were used, such as Visual Studio 2022 for C++ programming and Visual Studio Code for editing web components and integrating with JavaScript and WebAssembly. GitHub [11] was used for version control.

Communication with the supervisor was maintained constantly, both in regular in-person sessions and via the university email, allowing for doubts to be resolved, approaches to be discussed, and technical decisions to be consolidated.

Conclusions and future work

In this chapter, the conclusions from the completed project are presented and possible future lines of work are outlined. Firstly, the main obtained results are summarized and afterwards, potential improvements and extensions that could enhance the tool in future stages are proposed.

Conclusions

To begin with, one of the objectives of the project was to develop a tool capable of automatically solving the *Exact Number* and *Longest Word* challenges of the *Cifras y Letras* game show, using search algorithms and data structures. This objective has been successfully achieved, since functional implementations have been developed that reproduce and extend the contest's dynamics.

In relation to this objective, the behavior and performance of different algorithmic solutions have been analyzed, comparing various alternatives and identifying their advantages and limitations in terms of execution time, accuracy, and adaptability.

Additionally, the project has achieved another of its main objectives: bringing these implementations closer to the user through the development of a web interface. For this purpose, C++ code has been integrated with WebAssembly, allowing the algorithms to run directly in the browser without the need for additional installations or external configurations. This decision ensures accessibility and ease of use, factors that are key to the success of our project.

Regarding user perception, an effort was made to maintain an aesthetic similar to that of the television program, promoting familiarity and immersion in the environment. Nevertheless, the implementation allows entering numbers and conditions beyond those considered in the original format, as well as automatically checking the meaning of the words generated in the RAE dictionary.

In the project, the combination of algorithmic techniques with an accessible interface makes it possible to reproduce and carry out processes that are traditionally manual.

Likewise, the choice of C++ as the base language, together with compilation to WebAssembly, has made it possible to integrate high-performance solutions in web environments, opening the door to future applications in contexts other than the contest.

In short, the project has achieved its objectives: it has developed a functional and accessible tool to automatically solve the *Cifras y Letras* challenges, it has studied the performance of various alternatives, and it has developed a web interface for the implementation capable of transferring the solutions to an interactive environment.

Future Work

As potential future options arising throughout the development of the project, reference can be made to two areas in which work can be done: one dedicated to implementation and the analysis of different algorithms; and another related to the web interface.

First, in relation to a future analysis of different algorithms, various lines of improvement are proposed. Among them it is worth highlighting: the incorporation of new game modalities that include the remaining challenges of the contest, both in the mathematical and linguistic areas, which can be addressed through computational techniques; the expansion of the number challenge to allow longer numbers, both in the target number and in the available digits and the implementation of the letter challenge in other languages, thus expanding the scope of the application and adapting it to diverse linguistic contexts through the use of dictionaries and specific rules of each language.

On the other hand, regarding the web interface, several aspects of interest are identified for future development. Among them are included: creating a version in which users can select the available operations, for educational purposes, also allowing the restriction of values; implementing an online game mode that promotes interaction among users; developing a player ranking system that encourages competitiveness; and, finally, incorporating a collaborative space for algorithm comparison, in which different

developers can contrast their solution proposals and evaluate their performance in a common environment.

Bibliografía

[1] Chiffres & Lettres, «https://www.chiffresetlettre.com/fr,» [En línea].

[2] «Real Academia Española» [En línea]. Available: https://dle.rae.es/. [Último acceso: 10 Marzo 2025].

[3] «Cifras y Letras» RTVE, [En línea]. Available: https://www.rtve.es/play/videos/cifras-y-letras/. [Último acceso: Marzo 2025].

[4] J. Colard, «CaLao» [En línea]. Available: http://jacques.colard.free.fr/CaLao/. [Último acceso: 10 Marzo 2025].

[5] «Comptes & Mots» [En línea]. Available: https://www.comptes-mots.net/. [Último acceso: 10 Marzo 2025].

[6] R. S. Kevin Wayne, «Algorithms» de *Algorithms*.

[7] E. Fredkin, «Trie Memory» de *Communications of the ACM*, vol. 3.

[8] «Bootstrap» [En línea]. Available: https://getbootstrap.com/. [Último acceso: Marzo 2025].

[9] «Emscripten» [En línea]. Available: https://emscripten.org/. [Último acceso: 2025 Marzo].

[10] A. Haas, A. Rossberg, D. Schuff, B. Titzer, D. Gohman, L. Wagner, A. Zakai, J. Bastien y M. Holman, «Bringing the Web up to Speed with Web Assembly» de *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[11] «GitHub» [En línea]. Available: https://github.com/. [Último acceso: Marzo 2025].

[12] J.-M. Alliot y C. Vanaret, «(The Final) Countdown» de *Global Conference on Artficial Intelligence*, 2015.

[13] S. Colton, «Countdown Numbers Game: Solved, Analysed, Extended» de *Global Conference on Artificial Intelligence*, 2015.

[14] «Cifras y Letras 2» [En línea]. Available: https://play.google.com/store/apps/details?id=com.cadev.cifrasyletras2&pli=1. [Último acceso: Marzo 2025].

[15] «Cifras y Letras,» [En línea]. Available: https://play.google.com/store/apps/details?id=com.cadev.cifrasyletras.

[16] «Countdown Numbers and Anagrams,» [En línea]. Available: https://play.google.com/store/apps/details?id=uk.co.jackhurst.countdown.production. [Último acceso: Marzo 2025].

[17] «Chiffres Lettres and Duels» [En línea]. Available: https://play.google.com/store/apps/details?id=com.mawuvidevs.mots_et_comptes. [Último acceso: Marzo 2025].

[18] «Cifras y Letras» [En línea]. Available: https://www.epriego.net/educa/cifras-y-letras.asp.

[19] «Des Chiffres et des Lettres» [En línea]. Available: https://www.deschiffresetdeslettres.com/.

[20] V. K. Boo y P. Anthony, «A Data Structure Between Trie and List for Auto Completion» de *Knwoledge Technology Week 2011*, 2011.

[21] «Resolviendo Cifras y Letras» [En línea]. Available: https://eltamiz.com/elcedazo/series/resolviendo-cifras-y-letras/.

[22] «C++ reference» [En línea]. Available:
https://en.cppreference.com/w/c/language/object.html .