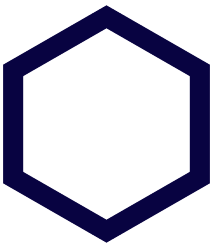
Universidad del País Vasco

Herramientas Avanzadas de Desarrollo de Software

Memoria Técnica

HADSagono

Nicolás Aguado, Asier Contreras y Martín Horsfield



16 de mayo de 2024

Índice

1. Resumen

El proyecto consiste en el desarrollo de un juego rompecabezas basado en hexágonos donde hay que juntar un mínimo de 3 hexágonos del mismo valor para fusionarlos en uno. Se puede jugar tanto una partida individual o una partida contra una IA. Mientras juegas la partida, una inteligencia artificial juega simultáneamente en un tablero inicial idéntico para determinar quién es mejor, si el humano o la IA. El juego finaliza cuando uno de los dos jugadores realiza un movimiento y en el nuevo tablero no hay movimientos legales posibles.

El frontend se ha desarrollado utilizando React y Vite, creando una interfaz de usuario interactiva y atractiva. Por otro lado, el backend se ha implementado con Node.js, creando una API que permite la comunicación entre el cliente y el servidor. Para el modo contra la IA, se han analizado los distintos modelos disponibles (a fecha de implementación de la aplicación) y se ha optado por un juego semi-*guiado*, en el que los modelos actuales no son todavía lo suficientemente capaces como para elaborar un razonamiento capaz. Entonces, se ofrecen alternativas y se le dan vías al modelo LLM para elegirlas.

Además, el juego tendrá un sistema de autoguardado de partida, que se almacena en una base de datos de MongoDB. No es necesario registrarse por lo que para almacenar las partidas el sistema se encargará de crear un código de partida único, tanto para las partidas individuales como para las partidas contra la IA. Mediante ese código, el usuario podrá cargar la partida y seguir con ella siempre que quiera.

Finalmente, el proyecto se ha dockerizado para facilitar su despliegue y ejecución en diferentes entornos (sin generar fallos, independientemente de la arquitectura), garantizando la portabilidad y consistencia del mismo.

2. Introducción y objetivos

2.1. Introducción

Este proyecto tiene como objetivo desarrollar un juego de rompecabezas *multi-dispositivo* basado en la unión de hexágonos con números del mismo valor. El jugador deberá identificar y combinar los hexágonos adyacentes que contengan el mismo valor, lo que agrupará la ruta de números seleccionada en un único hexágono y generará nuevos números de forma aleatoria en el tablero. El desafío radica en lograr la mayor puntuación posible a través de la creación de rutas y combinaciones válidas. Además, también estará disponible un modo *multijugador* en el cual se enfrentará al jugador frente a la inteligencia artificial en un desafío por ver quién consigue más puntos. En ambos casos, el juego terminará cuando el tablero no contenga movimientos posibles.

Figura 1: Modo un jugador del juego, en una partida recién empezada

2.2. Objetivos

Entre la gran variadad de objetivos y requisitado que tiene este proyecto, los más destacables son los siguientes:

1. Implementar un tablero de juego utilizando una representación en formato JSON, donde cada posición estará ocupada por un hexágono con un número.

2. Permitir que el usuario pueda interactuar con el tablero, seleccionando y uniendo los hexágonos adyacentes del mismo valor.

3. Desarrollar un algoritmo que genere nuevos números de forma aleatoria en el tablero después de cada combinación exitosa.

4. Integrar un módulo de inteligencia artificial, utilizando la biblioteca LangChain, que permita al jugador enfrentarse a un oponente basado en Inteligencia Artificial.

5. Implementar una infraestructura escalable y portable utilizando Docker, facilitando así la ejecución y despliegue del proyecto en diferentes entornos.

6. Utilizar las tecnologías de desarrollo de software más actuales, como VS Code, Node.js y React, para garantizar un desarrollo eficiente y una experiencia de usuario óptima.

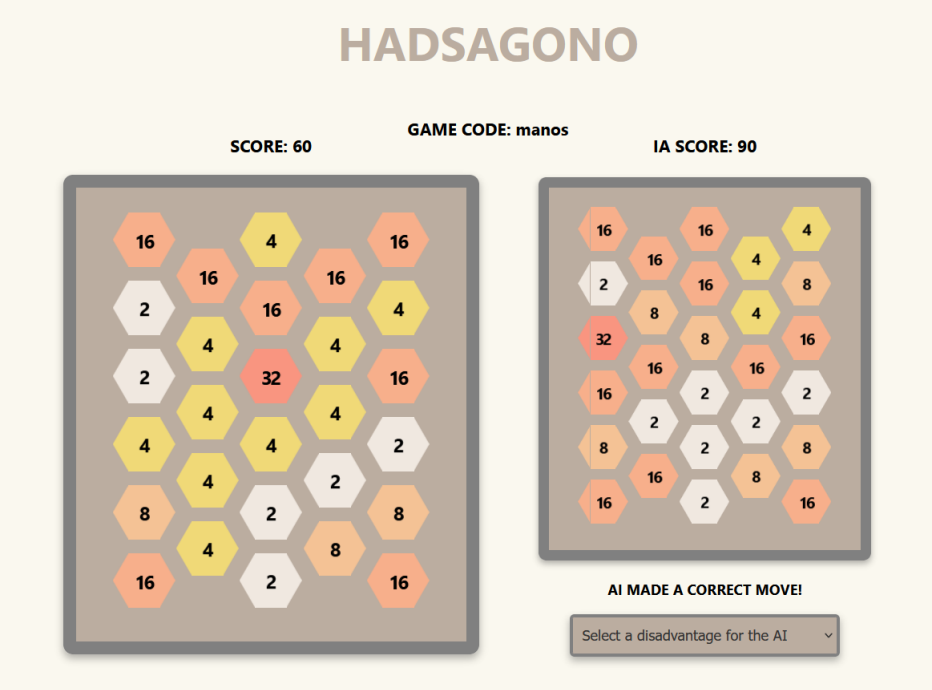


Figura 2: Modo multijugador vs IA del juego

3. Gestión del Proyecto

En cuanto a la gestión del proyecto de cara a la colaboración entre miembros del equipo, los plazos de realización de las tareas y la gestión de dependencias y comunicaciones se ha utilizado un **ciclo de vida ágil - iterativo**. Semana a semana, se han definido las tareas (escogidas de la lista común 3.2) a realizar para los miembros del equipo. El objetivo ha sido ir incorporando requisitos al proyecto, creando *versiones* intermedias funcionales de la aplicación. Así, poco a poco añadiendo requisitos, la aplicación ha ido tomando forma y gracias a una buena gestión se han identificado las dependencias entre tareas y salvaguardado los posibles riesgos.

3.1. Alcance del Proyecto

El alcance inicial era desarrollar un juego donde los hexágonos del mismo valor se fusionaban al unirse, mientras una inteligencia artificial jugaba simultáneamente en un tablero idéntico para determinar quien era mejor, si el humano o la IA. Sin embargo, a medida que avanzaban las semanas y nos familiarizábamos más con la tecnología, incorporamos nuevas funciona-

lidades que ampliaron significativamente el alcance original además de más requisitos que el mismo profesor introdujo durante las clases.

En cuanto al frontend, se ha desarrollado una interfaz de usuario interactiva y visualmente atractiva utilizando la biblioteca React y la implementación de Vite como herramienta de construcción.

El frontend no deberá de gestionar nada del sistema, solamente deberá de gestionar que mostrar en cada momento. Todas las tareas y funciones que hacen que el juego sea un juego y se pueda jugar se encargará el backend. Para ello, en el backend, se ha creado una API utilizando Node.js, que permite la comunicación entre el cliente y el servidor.

El juego implementa un sistema de autoguardado de partida para el juego. Esto permite almacenar y recuperar los estados de las partidas en los modos individual y contra la IA. Además, se han desarrollado funcionalidades adicionales, como la eliminación automática de partidas antiguas o con pocas jugadas, y la capacidad de realizar backups y restauraciones de los datos.

Un aspecto fundamental de este proyecto es la integración de la Inteligencia Artificial. Utilizando bibliotecas especializadas como LangChain o similares, se ha desarrollado un oponente basado en IA que puede jugar contra el usuario. Para aumentar la dificultad y proporcionar una experiencia más desafiante, se han ideado diferentes desventajas que el usuario puede aplicar a la IA durante la partida.

Finalmente, una vez finalizado el proyecto, se ha dockerizado para facilitar su despliegue y ejecución en diferentes entornos sin generar fallos. Esto asegura la portabilidad y la consistencia del proyecto en distintos sistemas.

En resumen, el alcance de este proyecto abarca el desarrollo de un juego de rompecabezas, la implementación de un backend robusto y escalable, la integración de un sistema de base de datos utilizando MongoDB, la incorporación de un oponente basado en Inteligencia Artificial y la dockerización del proyecto para facilitar su despliegue y ejecución.

3.2. Tareas del Proyecto

De cara a poder desarrollar el trabajo en grupo, se han dividido las tareas para el desarrollo en paquetes de estimación temporal similar. Aquí se enumeran las más destacables sin ningún órden en particular.

- Definición de estructura JSON común para comunicación entre cliente y servidor (jugadas, desventajas, movimientos, respuestas y tableros).
- Definición de rutas en el backend, junto con parámetros de entrada y salida para comunicación con el cliente.

- Lógica de negocio - Implementación de juego base mediante matrices con listas de movimientos posibles.
- Lógica de negocio - Definición de prompts e implementación de librería langchain para comunicación con IA.
- Lógica de negocio - Investigación sobre ahorro de tokens y optimización de recursos en la librería langchain.
- Implementación de capa de acceso a datos, basándose en instancia de mongodb y bajo el esquema común definido.
- Investigación y aprendizaje básico de libería *React*
- FrontEnd - Definición y estilado de *Tablero* básico, con coloreo de distintos números.
- FrontEnd - Implementación de estilado y llamadas al backend para hacer jugadas. Recepción de información y actualización del tablero.
- FrontEnd - Implementación de estilado y llamadas al backend para el modo contra la IA.
- FrontEnd - Definición de menú, tratamiento de errores en llamadas al backend.
- BackEnd - Implementación de desventajas según el esquema común.
- FrontEnd - Implementación de estilado y llamadas al backend para las desventajas según el esquema común.
- Deployment - Investigación de Dockerfiles y realización de empaquetado de aplicación.
- Deployment - Investigación de docker compose y realización de empaquetado completo de aplicación. Variables de entorno y base de datos.
- BackEnd - Condición de finalización de juego - Diseño e implementación de algoritmo para obtener jugadas posibles.

Una de las dificultades del proyecto ha radicado en identificar correctamente las dependencias entre tareas y la coordinación grupal mediante reuniones para poder producir un desarrollo constante en el proyecto. Además, se han repartido las tareas tal que si un miembro del equipo realizaba una tarea de la lista, éste era también el encargado de redactar su parte correspondiente en la memoria técnica (este documento).

4. Tecnologías

Como IDE se ha usado Visual Studio Code [7] junto a Copilot [6] para ayudar en la generación de código y para preguntar dudas y cuestiones relacionadas con el proyecto que hayan podido surgir. Además, como IA complementaria se ha usado el famoso asistente Claude [2] en su modelo más competitivo, Opus.

Para almacenar las partidas y la gestión de las propias partidas guardadas hemos usado una base de datos de MongoDB [8] ya que es la más adecuada para trabajar con JSON. Gracias a usar una BD el juego permite almacenar y recuperar los estados de las partidas, tanto en el modo individual como en el modo de juego contra la IA.

Como entorno de ejecución para el backend del proyecto se ha utilizado Node.js, junto a su gestor de paquetes por defecto npm [9]. Esta elección permite aprovechar las capacidades de JavaScript tanto en el frontend (con React [5] + Vite [12]) como en el backend (con Express [1]), facilitando la integración entre ambas partes de la aplicación.

4.1. Lenguajes de Programación

4.1.1. Lenguajes de Programación

Como lenguaje de programación principal para el proyecto se ha usado JavaScript. Con él, se han desarrollado toda la lógica del backend utilizando Node.js, como la interfaz de usuario del frontend, mediante la biblioteca React.

4.1.2. Librerías

Frontend:

- React: Es una biblioteca de JavaScript para la construcción de interfaces de usuario modernas y de forma más sencilla y rápida.
- React Icons: Una librería que proporciona un conjunto de iconos para mejorar la interfaz gráfica.
- Node Snackbar: Una biblioteca utilizada para mostrar notificaciones y mensajes de retroalimentación al usuario.
- Vite: Es una herramienta de construcción que optimiza el desarrollo y el rendimiento del frontend que se usa de forma predeterminada con React.

Backend:

- Node.js: Un entorno de ejecución de JavaScript que permite construir el backend de la aplicación.
- Express: Framework que proporciona el servidor web y la estructura base del backend.
- Mongoose: Una biblioteca que facilita la interacción con la base de datos MongoDB para la persistencia de datos.
- Dotenv: Una librería que permite cargar variables de entorno, como la URL de conexión a MongoDB o el key de Groq.
- Moment: Librería para trabajar con fechas y horas.
- LangChain (ChatGroq, ChatPromptTemplate): Esta es una librería de LangChain, la biblioteca de inteligencia artificial utilizada en el proyecto. ChatGroq proporciona funcionalidades para la generación de respuestas de chat utilizando modelos de lenguaje. ChatPromtTemplate ha sido usado para definir y manipular plantillas de prompts de chat.

4.2. Herramientas Software

Para este proyecto se han usado múltiples herramientas de software actuales, tanto herramientas que ya conociamos como algunas nuevas.

Al ser un proyecto colaborativo en el que más de una persona ha trabajado en el, debíamos de llevar un control de versiones adecuado. Para ello, se ha usado GitHub como plataforma de alojamiento de repositorios del proyecto para el control de versiones.

Gracias al paquete de estudiantes gratuito que ofrece GitHub, también hemos usado Copilot tanto la ayuda en la generación automática de código como el chat interno que ofrece. Además, también hemos usado Claude como LLM adicional.

Para el frontend, se ha usado Vite + React. Vite es un entorno de desarrollo rápido y ligero que permite crear aplicaciones web modernas con una configuración sencilla. Por otro lado, React es una biblioteca de JavaScript utilizada para construir interfaces de usuario interactivas y reactivas.

Para el desarrollo de backend del servidor, se ha usado Node.js en la versión v18.18.0 junto a npm en la versión 9.8.1. Además, se ha usado Express.js para facilitar la creación de las APIs.

Como base de datos se ha usado MongoDB en la versión 4.4.29 y Mongoose como biblioteca de modelado de objetos para interactuar con MongoDB. MongoDB es una base de datos NoSQL orientada a documentos, almacena los datos en BSON (Binary JSON) y en el juego trabajamos todo con JSONs por lo que es la herramienta de almacenamiento más adecuada para este proyecto.

Para el despliegue de la aplicación dentro de contenedores de software se ha usado Docker. Docker es una plataforma de contenerización que permite empaquetar una aplicación junto con todas sus dependencias en un contenedor aislado. Los contenedores de Docker proporcionan un entorno consistente y portátil para la ejecución de aplicaciones. Facilita la implementación y el despliegue de aplicaciones en diferentes entornos, como desarrollo, pruebas y producción. Además, permite la escalabilidad y la gestión eficiente de aplicaciones al aislar los servicios en contenedores independientes.

4.2.1. Entornos de Desarrollo Integrado

Como se ha comentado previamente, para el desarrollo se ha usado Visual Studio Code [7] junto a Github Copilot [6] y Github Copilot Chat.

5. Implementación

5.1. Frontend

El componente principal de la interfaz de usuario es *Board.jsx*, que centraliza toda la lógica y la interacción del juego. Este componente utiliza el estado de React para manejar diversas variables, como la selección de hexágonos, el código del juego, el modo de juego y la interacción del usuario. Los métodos del componente, como handleMouseDown, handleMouseEnter y handleInputChange, actualizan el estado y, en algunos casos, realizan solicitudes al backend a través de funciones como *postMoves* y *postMovesIA*.

La renderización de la interfaz de usuario se adapta dinámicamente al estado de la aplicación según el useState que esté en true en ese momento. Dependiendo de si se está mostrando el menú principal, si el usuario está jugando en modo individual o contra la IA, o si los datos del juego se han cargado correctamente, el componente Board se encarga de renderizar los elementos correspondientes.

Además, la clase *Board.jsx* se encarga de aplicar estilos y animaciones a los elementos de la interfaz, como los hexágonos y los botones. Esto se logra mediante el uso de un fichero CSS externo con el nombre *Board.css* al que se

le hace referencia y etiquetas de estilo internas en el propio *Board.jsx*

Por último, el frontend también se encarga de realizar comunicaciones con el backend a través de solicitudes HTTP utilizando fetch. Estas solicitudes se realizan a las rutas definidas en el servidor Node.js.

5.2. Lógica de Negocio - board.js

Para representar al juego en sí se ha decidido hacer uso de la orientación a objetos. Se ha creado la clase *Board* y ésta contendrá todos los métodos necesarios para representar al tablero, hacer movimientos y efectuar comprobaciones.

Lo primero a destacar es la manera de representación del juego. El tablero será un conjunto de hexágonos entre los que se podrá mover uno dependiendo por sus lados. Pero claro, hay que transformar esos hexágonos y movimientos en algo fácilmente manipulable y compatible con los lenguajes de programación. Se ha optado por la representación del tablero como una **matriz** de columnas impares y de filas pares con algunas casillas deshabilitadas de la última fila. Esta figura será fácilmente manipulable y escalable para todo tipo de algoritmos. Mediante unas sencillas reglas es posible generar los movimientos posibles de nodo a nodo (de casilla a casilla de la matriz) sea cual sea el tamaño de ésta. (Figura 3).

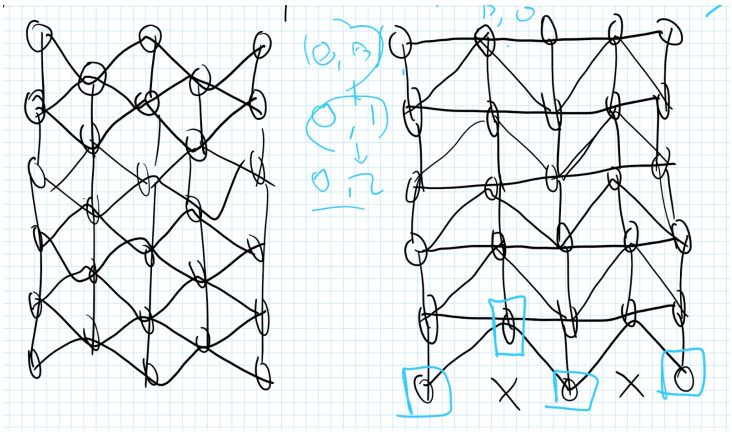


Figura 3: Boceto inicial de la representación de hexágonos como matriz.

Entonces, una vez se tiene la matriz de movimientos y los movimientos posibles entre casillas, se pueden representar unas *acciones* mediante una *casilla de salida* y las direcciones a recorrer, casilla por casilla. Por ejemplo, para empezar en la casilla `[0,3]` y moverse abajo a la derecha y después abajo, se representaría con el json `{"moves": [[0,3], [1, 1], [1, 0]]}`.

Como ya se saben los movimientos posibles desde cada casilla, la verificación de si una *acción* es válida o no se realiza de forma trivial. Después, se borran todas las casillas afectadas, se recalculan los nuevos valores y se actualiza la puntuación del jugador.

5.3. Base de Datos - MongoDB y dataAccess.js

5.3.1. Diseño de la BD

La base de datos tiene varios esquemas. Vamos a describirlos a continuación:

- **Board**: Es el esquema del juego. Almacena el *tablero*, la *puntuación*, el *número de movimientos*, el *código de identificación* y la *fecha*. MongoDB añade otros datos como `_id`.

```
const boardSchema = new Schema({
board: [],
score: Number,
movecount: Number,
code: String,
date: { type: Date, default: Date.now }
});
```

- **BoardAI**: Es el esquema del juego en multijugador (con la IA). Almacena lo anterior y varios datos adicionales.

```
const boardAISchema = new Schema({
board: [],
score: Number,
movecount: Number,
code: String,
ai: {type: Boolean, default: false},
aiBoard: [{type: [], default: []}],
aiScore : {type: Number, default: 0},
consumedDisadvantages : [{type: [], default: []}],
date: { type: Date, default: Date.now }
});
```

- **BackupBoard** y **BackupBoardAI**: Son dos esquemas idénticos a los anteriores respectivamente. Sirven como copias de seguridad para la base de datos, ya que tenemos funciones con la capacidad de borrado

y cualquier problema podría llevar a la eliminación de todos los datos del almacenados.

5.3.2. dataAccess.js

Tenemos una clase estática de JavaScript que actúa como nuestro acceso a datos para la BD de MongoDB. La clase implementa varias funciones estáticas para acceder y almacenar datos, y otras funciones auxiliares.

Todas las funciones del dataAccess dependen de la función `codeExists(code)`. Que dado un código, buscará en qué colección está (*Board o BoardAI*). Además de eso, si el código existe en ambas colecciones (un error), llamará a otra función (`purgeCode(code)`) que se encarga de tratar el problema.

Esto es un ejemplo de la mantenibilidad automática de nuestra base de datos. La intervención directa del gestor de base de datos solo es necesaria en casos de fallos directos en la BD.

Otra función interesante es `generateCode()`. Nuestra aplicación, al **no** tener usuario y contraseña, almacena los documentos con un **código de 5 caracteres**. Para que el código sea más legible, llamamos a una API ('`random-word-api.herokuapp.com`') que genera una palabra en inglés de 5 letras.

Este código es nuestra "**clave primaria**", y lo escribimos entre comillas porque Mongo no lo considera directamente como una clave primaria. Pero esta clave será única así que sirve de manera identificativa para el resto de funciones de dataAcess.js.

La función `save()` se llama cada tres movimientos del usuario. Esto asegura que cuando la BD esté separada del servidor de *back-end*, las comunicaciones no sean muy frecuentes para evitar problemas de rendimiento.

La función `cleanup()` se llama cada vez que se crea un nuevo juego, ya sea de un jugador o multijugador. La función elimina todas los juegos con **menos de tres movimientos** y todas las tablas con **más de tres meses de antigüedad**. En la base de datos no debería de haber juegos con menos de tres movimientos como acabamos de mencionar, pero lo revisamos como redundancia.

Para realizar pruebas, hemos utilizado un servicio de MongoDB alojado en `martinh.info:27017`. Al tener un procesador virtualizado, se tuvo

que realizar varios apaños para poder alojarlo correctamente y debido a ello, se utilizó una versión de MongoDB anterior (4.4.29). Ver el apartado 'docker' para saber como lo hemos gestionado en producción.5.7

5.4. Gestión de Rutas con Express

En el desarrollo del back-end se ha utilizado el framework *Express* dentro del lenguaje Node.js, siguiendo una arquitectura basada en routers. Cada router agrupa un conjunto de rutas relacionadas y proporciona una separación entre la lógica de negocio y el enrutador.

Se han creado tres enrutadores:

- **singleplayer.js** - Principal controlador del modo *un jugador*. Contendrá las rutas para crear una nueva partida y para hacer movimientos. Utilizará la lógica de negocio y hará de intermediario entre el cliente y la base de datos.
- **multiplayer.js** - Principal controlador del modo *contra la IA*. Contendrá rutas parecidas a las del modo *ün jugador*, además de las necesarias para poder comunicarse con la IA y para poder aplicar desventajas.
- **load.js** - Principal encargado de *restaurar* las partidas guardadas en la base de datos. Se comunicará con la capa de acceso a datos.

Estos tres enrutadores tendrán, además, que establecer correctamente las variables de sesiones de usuario, hacer copias de seguridad en la base de datos, controlar los códigos de error de las librerías y de la base de datos y unos cuantos detalles más.

En cuanto a cómo se guarda un juego, se ha definido un formato *JSON* que se guardará en la variable de sesión del usuario. De esta forma, el servidor tendrá una visión en todo momento de la partida del usuario, pero teniendo el inconveniente de que el usuario no podrá jugar dos partidas a la vez.

JSON enviado desde el servidor para un movimiento en multijugador.

```
let resjson = {
    "board": b.board,
    "iaboard": bAI.board,
    "score": b.score,
    "iascore": bAI.score,
    "iaPathGeneration" : parsedData.
       iaPathGeneration,
```

```
        "consumedDisadvantages": parsedData.
            consumedDisadvantages,
        "iaResult" : aiResu
    };
```

Para contrarrestar el inconveniente, cada 3 movimientos se guardará el juego en la base de datos. Se generará un código y se le proporcionará al usuario para que pueda restaurar la partida en un estado intermedio.

5.5. Uso de LLMs

Se nos propuso utilizar modelos LLM para jugar a nuestro juego. Muy pronto nos dimos cuenta que estos modelos no están hechos para este tipo de actividades y no podían procesar la información que le estábamos proporcionando. Decidimos optar por "guiarle" a la IA y darle los datos más procesados de antemano para que su decisión no fuese tan complicada.

5.5.1. Algoritmo de búsqueda de caminos

Decidimos darle a la IA los movimientos posibles directamente y que, **con contexto**, eligiese el camino más óptimo para ganar la partida.

El algoritmo está contenido en la función `findFirstPath()` que recibe el *tablero*, el *array de movimientos*, la *fila*, la *columna*, el *camino* y la *longitud necesaria*. La fila, la columna y el camino son necesarios para la recursividad pero al comienzo se llamarán con [0, 0, []], es decir, la fila inicial, la columna inicial y un camino vacío. Estos valores cambiarán a medida que la función entre en la matriz. Esta función devuelve el primer camino de la longitud pedida en esa fila y columna. Si no se encuentra ningún camino, devuelve `null`.

Para el resto de filas y columnas tenemos `findSolutions` que acepta el *tablero*, la *matriz de movimientos* y la *longitud necesaria*. Esta función recorre **la matriz entera**, buscando caminos de la longitud deseada para cada coordenada de la matriz. Si no encuentra solución para esa longitud, **empezará la búsqueda otra vez** con `length-1`, es decir, una longitud menor. Esto se repetirá hasta que se encuentre al menos un camino de longitud mayor a 3. Si no hay caminos de longitud mayor o igual a 3, significa que ya no hay más movimientos posibles. Esta función se usa también para comprobar si quedan más movimientos o acabar la partida.

Con estas funciones (y otras varias auxiliares), le podemos proporcionar a la IA caminos con longitud a medida. Y elegirá una solución.

5.5.2. Contexto y optimización de tokens

En vez de pasar individualmente el contexto para cada pregunta, le pasamos el contexto al comienzo del 'chain' con un prompt inicial (que se iniciará cuando se inicialice el back-end). El prompt base es el siguiente:

"You are playing a connecting numbers game. The numbers are represented in a matrix. You have to connect a minimum of 3 numbers of the same value. You are restricted to only some possible movements along the matrix. I will pass on an array of possible movements like this: ï,j": [[a,b], [c,d]], ... where i is the row of the matrix and j is the column of the matrix. From i,j you can move in the directions displayed by [a,b] or where a is the difference in the rows"(i variable) and b is the difference in the columns (j variable) You must connect the most amount of numbers. Now, please choose one of the possible plays I will pass you and return the number that you selected. Only respond with the number of the play you want to make. Do not explicitly include the play number. The response must be 1 digit long. Do not be verbose. Do not provide any explanations."

Esto evita usar tokens para cada movimiento (consulta que le hagamos). Entonces, cuando haya que hacer un movimiento, se le pasará la matriz, la matriz de movimientos y los caminos posibles tal cual.

Utilizamos varios modelos de IA gestionados por Groq, y utilizamos Langchain como interfaz para las peticiones. Groq tiene un límite de 'tokens' 4.

'En el contexto de los grandes modelos de lenguaje (LLM, por sus siglas en inglés), los tokens son las unidades de texto más pequeñas que el modelo procesa y genera. La tokenización es el proceso de dividir el texto de entrada en estas unidades más pequeñas.' *Claude 3 Opus*

ID	REQUESTS PER MINUTE	REQUESTS PER DAY	TOKENS PER MINUTE
gemma-7b-it	30	14,400	15,000
mixtral-8×7b-32768	30	14,400	5,000
llama3-70b-8192	30	14,400	6,000
llama3-8b-8192	30	14,400	30,000

Figura 4: Limites de tokens que establece Groq

Nosotros utilizamos `llama3-70b-8192` y `gemma-7b-it`.

Cuando la IA llega al límite de tokens, Langchain no devuelve un error, sino que espera a que haya tokens o peticiones libres para continuar. Al usuario le aparece una pantalla de espera.

5.5.3. Sistema de desventajas

Tenemos implementado un **sistema de desventajas** ya que la IA juega *demasiado* bien con el algoritmo (o las guías) que le proporcionamos. Una de las desventajas es **degradar el modelo** que utiliza. Con esta función, se puede pasar automáticamente del modelo `70b de Llama3` al `7b de Gemma`. Una consecuencia es que la utilización de tokens se distribuye por modelos.

Las otras dos desventajas también afectan a la IA. Una de ellas es la **limitación de la información** proporcionada. Para el resto de la partida, la IA solo podrá ver las primeras 3 filas y 3 columnas, pero no sabrá lo que hay en el resto del tablero. Las decisiones que tomará serán válidas pero a ciegas, y la partida podrá acabar por falta de movimientos si no lo gestiona bien. La última desventaja es la **limitación de la longitud de los caminos** proporcionados. En esta, la IA solo podrá realizar movimientos con una longitud acortada, limitando la puntuación que puede obtener.

Esta es la configuración inicial del chat:

```
const model = new ChatGroq({
    apiKey: process.env.GROQ_KEY_1,
    model: "llama3-70b-8192",
  });
  const model2 = new ChatGroq({
    apiKey: process.env.GROQ_KEY_1,
    model: "gemma-7b-it",
  });

const prompt = ChatPromptTemplate.fromMessages([
    ["system", aitools.generateBasePrompt()],
    ["human", "{input}"],
]);
```

Esta es la función que elige el modelo LLM:

```
function getChain(req) {
    if (req...consumedDisadvantages.includes(3)) {
        return prompt.pipe(model2);
    }
```

```
    return prompt.pipe(model);
}
```

5.6. Códigos de Error

Durante todo el proyecto se ha usado un sistema de *códigos de error* para tratar mejor el intercambio de información entre capas y siempre obtener una visión concisa de lo que estaba pasando cuando algo iba (o va) mal. El documento se encuentra en la carpeta de *docs/* dentro del proyecto.

Error	Descripción
257	Movimiento ilegal (Dos números distintos)
259	No quedan más movimientos para la IA
282	Error petición API de generación de código
284	Error al hacer 'load' desde la bd. El código existe tanto para single como para multiplayer. Se borrarán ambos.
302	La IA ha sido incapaz de escoger un número de los que se le han pasado, o no lo ha devuelto de la manera correcta
100	Todo ha ido bien

Extracto del documento codigoserror.txt

5.7. Docker

Como manera de compilación y ejecución del proyecto en producción, se ha usado *Docker [3]* como plataforma todo-en-uno. Se ha optado por usar dos contenedores conectados entre sí por el fichero *docker-compose.yml*

En cuanto al primer contenedor, se ha decicido realizar una integración con el backend y el frontend. El *back* es una aplicación express que tiene que escuchar en un puerto, es un servidor web. En cambio, el *front* en React al ser compilado se transforma en unos ficheros html, css y javascript. En una primera aproximación se pensó en tener dos servidores web, uno para servir los archivos del *front* y otro para servir los archivos del *back*. Siguiendo

el principio *KISS*[11] se optó por hacerlo más sencillo y combinarlo todo en un único servidor web. Entonces, en el Dockerfile se compila primero el frontend y luego se copia a una carpeta especial del backend donde el servidor express *servirá* estos archivos estáticos.

Dockerfile-frontback

```
# Stage 1: Build the React application
FROM node:20-alpine AS build

WORKDIR /app
COPY ./front ./
ARG VITE_BACKEND_URL
ENV VITE_BACKEND_URL=${VITE_BACKEND_URL}
RUN npm install --save
RUN npm run build

# Stage 2: Serve back-end with Express
FROM node:20-alpine

WORKDIR /app

COPY --from=build /app/dist ./public

COPY ./back ./
RUN npm install --omit=dev

# Start the Express application
CMD ["npm", "run", "prod"]
```

El segundo contenedor se refiere a la base de datos MongoDB. MongoDB es un SGBD excelente, pero su puesta en marcha requiere una configuración y conocimiento previo extensos. En primer lugar, para poder crear un usuario y una contraseña en la imagen de MongoDB de Docker [4] hace falta establecer unas variables de entorno y pasarle un *script* de creación de usuarioreafirmando (sí, reafirmando) esas mismas variables de entorno que se han declarado.

Script inicial init-mongo.sh

```
#!/bin/bash
mongo <<EOF
use hadsagono

db.createUser({
  user: "${MONGO_INITDB_ROOT_USERNAME}",
  pwd: "${MONGO_INITDB_ROOT_PASSWORD}",
  roles: [{
    role: 'readWrite',
    db: 'hadsagono'
  }]
});
```

```
EOF
```

Además, una vez se ha conseguido crear un usuario y una contraseña, dependiendo de la arquitectura que se utilice, se puede encontrar un problema con la CPU en entornos de virtualización [10]. Un VPS con una CPU virtualizada sin AVX con Proxmox es justo el entorno en el que se ha desplegado la aplicación (6.2), lo cual ha implicado adaptar el programa entero (capa de acceso a datos y docker) para poder usar una versión antigua (pero todavía soportada) de MongoDB.

Por último, a modo de protección, se ha decidido no exponer la base de datos en el propio servidor (no redirigir los puertos) y depender únicamente de las redes internas de Docker para la comunicación. Entonces, la **única** manera de acceder a MongoDB es pasando por la aplicación desplegada.

Con todos estos ajustes, el *docker-compose.yml* ha quedado así:

.

```
services:
  back-front-express:
    build:
      dockerfile: Dockerfile-frontback
    ports:
      - "${PORT}:${PORT}"
    environment:
      - MONGO_URI=mongodb://${MONGO_INITDB_ROOT_USERNAME}:${
          MONGO_INITDB_ROOT_PASSWORD}@database:27017/hadsagono
      - GROQ_KEY_1=${GROQ_KEY_1}
      - PORT=${PORT}
    depends_on:
      - database
    restart: unless-stopped
    networks:
      - public
      - internal

  database:
    image: mongo:4.4.29
    volumes:
      - ./mongodata:/data/db
      - ./db/init-mongo.sh:/docker-entrypoint-initdb.d/init-
          mongo.sh
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_DATABASE: hadsagono
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_INITDB_ROOT_USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_INITDB_ROOT_PASSWORD}
    command: mongod --auth --port 27017
    restart: unless-stopped
    networks:
      - internal
```

```
networks:
  public:
    driver: bridge
    internal: false
  internal:
    driver: bridge
    internal: true
```

6. Tareas adicionales

6.1. Singleplayer

El modo *un jugador* para nosotros no fué un extra a añadir, sino una base desde donde partir. Empezamos realizando esta parte para obtener y conseguir toda la lógica necesaria para tener un juego *que funcionara*, para luego implementar el modo multijugador *vs IA*.

6.2. Aplicación pública en internet

Realizando unas modificaciones (introducidas en el apartado de Docker 5.7) y siguiendo la guía de *self-hosting* C.1 se ha abierto una instancia de la aplicación en internet, bajo el dominio hads.nico.eus . Esta ejecución se encuentra en una máquina virtualizada VPS (sin soporte para AVX) en Núremberg (Alemania), bajo el proveedor de servicios netcup, con 4 vCores, 4GB de RAM y con una tarjeta de red de 1Gb.

6.3. Compatibilidad con dispositivos móviles

Como estaba subido en el sitio web mencionado con anterioridad, se decidió por invertir un poco de tiempo en añadir esta nueva funcionalidad si se veía factible y no generaba un aumento de tiempo excesivo. Para ello, hubo que realizar nuevos eventos para los gestos que se pueden realizar con la pantalla táctil, ya que los eventos de ratón no sirven para pantallas táctiles. Por lo que además de los eventos *onMouseDown*, *onMouseEnter* y *onMouseUp* hubo que realizar *onTouchStart*, *onTouchMove* y *onTouchEnd* tanto para singleplayer como para cuando se jugaba contra la IA. Además, hubo que realizar pequeñas modificaciones en el CSS para cuando el tamaño de la pantalla era inferior a 768px.

7. Dedicación Horaria

En cuanto a la dedicación horaria al proyecto, desde el equipo de desarrollo se ha intentado acotar el alcance del proyecto y no implementar demasiadas características *extra* para poder llegar a la estimación del proyecto de $\sim$ 50 horas por persona. Se detalla en el cuadro 1 la dedicación acumulada y por persona real del trabajo de realización del proyecto. Por *horas dentro y fuera de clase* se refiere al trabajo de implementación, mientras que también se expone un apartado para las horas de redacción del informe

Miembro	h Dentro de Clase	h Fuera de Clase	h Informe	Total
Nicolás	12	43	5	60
Asier	13	40	5	58
Martín	14	39	4	57
Total	39	122	14	175

Cuadro 1: Dedicacición Horaria, en Horas, por Persona

Como se puede observar, se ha superado el objetivo de 50 horas de dedicación, pero creemos que las horas extra invertidas han supuesto una mejora muy sustancial en la calidad del proyecto. Ese tiempo *extra* invertido ha sido para poder arreglar detalles y perfeccionar la jugabilidad y accesibilidad del juego.

Además, se ha sobreestimado la escritura del informe y se han dedicado más horas de las estimadas.

8. Conclusiones

8.1. Reflexión y conclusiones sobre el trabajo

En primer lugar, tenemos que destacar el uso de tecnologías punteras y actuales para implementar tanto el frontend (React y Vite) como el backend (Node.js, Express, MongoDB). Además, la elección de una arquitectura desacoplada, con una API que comunica entre cliente y servidor, da mucha flexibilidad al sistema.

Nos ha parecido que la integración de modelos de lenguaje (LLMs) en el proyecto para implementar un oponente IA es un aspecto muy novedoso. A pesar de las limitaciones actuales de los LLMs para procesar las reglas del juego directamente, proporcionar movimientos ya calculados entre los que la IA debe decidir, ha sido una solución eficiente. Nos imaginamos que en un futuro, una IA podría jugar a este juego sin ayuda humana.

Estamos orgullosos de haber desarrollado un juego accesible y compatible con dispositivos móviles. Además, el sistema de desventajas hace que el juego no sea tan repetitivo a largo plazo, y mantiene al usuario entretenido en todo momento.

La gestión del proyecto ha sido adecuada. Con un enfoque ágil-iterativo, nos hemos asegurado de que todos los cambios en los requisitos y funcionalidades adicionales se hayan implementado sin dificultades. La desviación horaria ha afectado a nuestro proyecto de manera positiva, permitiéndonos mejorar y arreglar funcionalidades.

Creemos que HADSagono es un proyecto muy completo y demuestra de manera práctica los conocimientos aprendidos tanto en la asignatura de HADS, como en otras asignaturas de la carrera. Al estar implementado de manera muy modular, creemos que está aplicación tiene mucho potencial en el futuro 8.3, si se quisiera expandir.

8.2. Lecciones Aprendidas

Para el desarrollo del frontend se ha utilizado Vite y React. Ha sido la primera vez que se ha trabajado con estas herramientas y gracias a los LLM y su asistencia la curva de aprendizaje ha sido menor y más rápida.

Hemos aprendido que las LLMs tienen usos muy particulares. Aunque la IA haya sido una gran ayuda para entender nuevas herramientas, todavía falta capacidad de razonamiento para razonar algunas cosas o jugar a juegos.

Realizar el proyecto ha reforzado la lección de la modularidad del código. Pasar el juego de un solo jugador a multijugador no fue un salto tan grande como pensábamos y pone en valor las buenas prácticas aprendidas durante la formación.

8.3. Mejoras a Futuro

De cara a realizar un trabajo similar en un futuro, se destacaría la importancia de revisitar la investigación del estado actual de los modelos LLM. A fecha de redacción de esta memoria, no se ha podido pasarle el juego (junto con sus reglas) *tal cual* a ningún modelo LLM y que éste produciese una secuencia válida de movimientos. Se ha tenido que *guiar* de cierta forma dando posibles soluciones al LLM. Aún así, dada la rápida evolución de los LLMs, se cree posible que en un futuro esta tarea sí pueda ser realizada y completada exitosamente, ahorrando tiempo de cómputo en el backend y produciendo jugadas menos predecibles.

Además, en un futuro se podría revisitar el algoritmo de generación de números, para implementar unos *niveles de dificultad*. Se podría implementar una progresión en la que el jugador va adquiriendo más y más experiencia en el juego según se van haciendo más difíciles los niveles.

Como posible expansión del juego también cabría considerar un modo *trampas* en el cual el cliente pudiese calcular la mejor jugada posible dado un tablero y se la mostrase al jugador.

Además, en cuanto al diseño visual o la jugabilidad, siempre hay aspectos en los que el juego puede mejorar añadiendo nuevos estilos o nuevos modos de juego. como ejemplo, a futuro se pueden añadir tableros más grandes, o que el usuario elija un tamaño (17*18 por ejemplo), ya que de la manera que implementamos el código es posible tanto en el backend con la generación de hexágonos o la verificación de movimientos legales, y en el aspecto visual del frontend.

En todo caso, las posibilidades de expansión son infinitas y el propio equipo de desarrollo tienen bastantes ideas en caso de que el proyecto sea revisitado.

Referencias

[1] Express Web Application Framework. https://expressjs.com/, [Online; accessed 15-May-2023]

[2] Anthropic: Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family, [Online; accessed 15-May-2024]

[3] Docker: Docker. https://docs.docker.com/, [Online; accessed 15-May-2024]

[4] the Docker Community: Official mongodb docker image. https://hub.docker.com/_/mongo/, [Online; accessed 15-May-2024]

[5] Facebook: React framework. https://react.dev, [Online; accessed 15-May-2024]

[6] Microsoft: Github Copilot in VSCode. https://code.visualstudio.com/docs/copilot/overview, [Online; accessed 15-May-2024]

[7] Microsoft: Visual Studio Code. https://code.visualstudio.com/, [Online; accessed 15-May-2024]

[8] MongoDB: Mongodb community server database. https://www.mongodb.com/try/download/community, [Online; accessed 15-May-2024]

[9] npmInc: node package manager. https://www.npmjs.com/, [Online; accessed 21-November-2023]

[10] Pandey, M.: Stack Overflow - Docker - MongoDB 5.0+ requires a CPU with AVX support. https://stackoverflow.com/questions/76126384/mongodb-5-0-requires-a-cpu-with-avx-support-container-failed-to-start, [Online; accessed 15-May-2024]

[11] Wikipedia: Kiss principle. https://en.wikipedia.org/wiki/KISS_principle, [Online; accessed 15-May-2024]

[12] You, E.: Vite packager. https://vitejs.dev/, [Online; accessed 15-May-2024]

A. Enlaces

El código fuente de la aplicación se encuentra desplegado en el repositorio de github martinh13/hadsagono.

Además, una instancia *demo* de la aplicación se encuentra desplegada en la dirección web hads.nico.eus.

Para facilitar el desarrollo en vscode, se encuentra disponible un paquete de extensiones en el siguiente enlace o buscando `hadsagono` en la pestaña de extensiones del propio visual studio code.

B. Ejecución en Entorno de Desarrollo

Instrucciones necesarias para ejecutar y modificar el proyecto en un entorno de desarrollo.

B.1. Frontend

Todo el desarrollo relacionado con esta sección ocurre en la carpeta *front*. El framework que se está utilizando es Vite-React (JavaScript).

- Primero, se necesitará abrir una terminal y ejecutar el siguiente comando para instalar todas las dependencias necesarias.

/front/

```
npm install --save
```

- Los archivos fuente del frontend de React se encuentran en la carpeta "front/src".
- Para ejecutar el servidor de desarrollo, deberás ingresar en la terminal.

/front/

```
npm run dev
```

B.2. Backend

Para el backend, la aplicación se basa en una base de datos mongodb, el framework *express* y en llamadas a la API para del proveedor de LLMs *Groq*.

1. Primero, habrá que abrir una terminal y ejecutar el siguiente comando para instalar todas las dependencias de node

/back/

```
npm install --save
```

2. Luego, habrá que renombrar el archivo ".env.example.ª ".env".

3. Se seberá después obtener una clave API del proveedor de LLMs *Groq* y se deberá insertar en el archivo ".env".

4. La aplicación también necesitará una base de datos *mongodb* para guardar las partidas. La base de datos **deberá** tener una base de datos creada llamada "hadsagono". La aplicación creará por sí misma tres colecciones: "boards", "boardaisz "backupboards", así que habrá que asegurarse de que no haya colisiones de nombres. Se deberá proporcionar una **URI de conexión** de mongo en el archivo ".env". Si se encuentra disponible una instancia de mongodb a mano para el entorno de desarrollo, se puede self-hostear una en la máquina local.

B.3. IDE Recomendado

Se puede usar el IDE que se desee para desarrollar el proyecto ya que la aplicación en sí no necesita ningún entorno de desarrollo *sofisticado* u personalizado en un sentido específico. Se trata de un desarrollo javascript para que el que es necesario abrir algunas terminales. Sin embargo, nosotros hemos utilizado y recomendamos utilizar visual studio code [7] para facilitar y hacer la experiencia de usuario mucho más cómoda. De cara a poder seguir la recomendación, hemos publicado nuestras extensiones recomendadas para vscode para trabajar en el proyecto en un paquete de extensiones en la marketplace de visualstudio. De esta manera, con un solo clic ya se encontraría el proyecto listo para su desarrollo.

C. Deployment en Producción

Instrucciones necesarias para compilar y desplegar el proyecto en un entorno de desarrollo.

C.1. *Self-Hosting*

Esta aplicación está diseñada para ser *construida y ejecutada* a través de docker. No se puede proporcionar una imagen de docker *preconstruida*porque algunas claves de API y otros parámetros de configuración deben ajustarse de antemano. Se recomienda tener al menos 3GB de espacio de almacenamiento en el servidor y se asume cierto conocimiento de administración de sistemas. Estos son los pasos para el autohospedaje de *hadsagono*:

1. Primero, se deberá clonar el repositorio:

@bash

```
git clone https://github.com/Martinh13/HADSagono.git
```

2. Para después hacer un *cd* a la carpeta clonada.

3. Luego, cambia el nombre del archivo ".env.example.ª ".env".

4. Abre el archivo ".envçon tu editor de texto favorito. Debes proporcionar una clave de API del proveedor de LLMs groq para la función de partidas multijugador. También puedes configurar el puerto de la aplicación web expuesta y el nombre de usuario y la contraseña de la base de datos mongodb que se generará.
5. Habrá que instalar el https://docs.docker.com/engine/install/ en el servidor si aún no está disponible.
6. Cuando estén todas las variables de entorno configuradas correctamente en el fichero ".env", simplemente se puede ejecutar

/hadsagono/ : @bash

```
docker-compose up -d
```

7. ¡Ahora la aplicación estará ejecutándose en el puerto que se ha especificado!

C.2. Extras Recomendados

- Si se está desplegando la aplicación en un servidor, lo más probable es que se quiera que la aplicación persista y se inicie automáticamente cuando se reinicie la máquina. En caso de que se esté ejecutando Linux y se gestione los procesos con *systemctl*, simplemente se puede ejecutar

@bash

```
sudo systemctl enable docker
```

- Se recomienda que se ejecute la aplicación web detrás de un *reverse proxy* (como nginx) para que se pueda proteger la aplicación a través de TLS/SSL y para tener más control sobre el servidor web. Un ejemplo de configuración de dominio de nginx para una aplicación que se ejecuta en el puerto 3456 en una IP *[MACHINE_IP]* y con un nombre de host *[DOMAIN]* podría ser:

/etc/nginx/conf.d/domains/[DOMAIN].ssl.conf

```
server {
listen       [MACHINE_IP]:443 ssl;
server_name [DOMAIN] ;
error_log    /path/to/your/error.log error;

ssl_certificate      /path/to/ssl/cert.pem;
ssl_certificate_key /path/to/ssl/cert.key;
ssl_stapling         on;
ssl_stapling_verify on;

# TLS 1.3 0-RTT anti-replay
if ($anti_replay = 307) { return 307 https://
    $host$request_uri; }
if ($anti_replay = 425) { return 425; }
```

```
        location ~ /\.(?!well-known\/|file) {
                deny all;
                return 404;
        }

        location / {
                proxy_pass http://127.0.0.0:3456;
        }

        location /error/ {
                alias /wherever/you/have/your/
                   document_errors/;
        }

        proxy_hide_header Upgrade;

        include /path/to/your/nginx.ssl.conf_*;
}
```

(Link al archivo en pastebin)