Solución del “Puzzle 15” de Tamaños Variables Mediante B&B

Alejandro Rodríguez, Rubén Palmer y Sergi Mayol

Resumen— Esta aplicación presenta una interfaz gráfica de usuario que le permite interactivamente un puzzle NxN para el juego del puzzle 15 basado en un algoritmo de ramificación y poda usando el patrón de diseño MVC.

Vídeo - ver vídeo

1. Introducción

En este artículo explicaremos el funcionamiento y la implementación de nuestra práctica estructurando y dividiendo sus partes en los siguientes apartados:

Inicialmente, se citarán las librerías usadas en este proyecto. Para aquellas que se hayan desarrollado específicamente, se expondrá una breve guía de su funcionamiento y uso.

Seguidamente, se explicará cuál es la arquitectura, interfaces, funcionamiento e implementación de nuestro **M**odelo-**V**ista-**C**ontrolador (MVC) así como los métodos de mayor importancia. Para facilitar la legibilidad, se separará en cada uno de los grandes bloques de la estructura; en este caso Modelo, Vista, Controlador y Hub

Finalmente, se mostrará una breve guía de cómo usar nuestra aplicación y un ejemplo de esta.

2. Librerías

En esta sección se explicarán las librerías implementadas y usadas para llevar a cabo el desarrollo de las prácticas y permitir la reutilización de las mismas durante el transcurso de la asignatura.

El principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo “overhead” al rendimiento del programa.

2.1. Better swing

Better swing [1] es una librería derivada de Java Swing que permite un desarrollo más amigable y sencillo de interfaces gráficas, al estar inspirado en el desarrollo web (HTML y CSS), concretamente, en el framework de “Bootstrap”. Se basa en “JfreeChart” para pintar las gráficas de líneas creando “wrappers” para facilitar su uso y manejabilidad.

2.1.1. Funcionamiento

El funcionamiento del paquete es muy sencillo, básicamente, la idea es crear una o varias ventanas con una configuración deseada y acto seguido ir creando y añadiendo secciones (componentes), donde posteriormente se pueden ir borrando y actualizando los componentes individual o grupalmente. Por ello, este paquete proporciona una serie de métodos para la ventana y las secciones.

¿Cómo funciona la ventana?

Para hacer que la ventana funcione es necesario crear una instancia del objeto `Window`, inicializar la configuración de la misma empleando el método `initConfig` recibiendo

por argumento la ruta donde se encuentra el archivo. En caso contrario se empleará la predeterminada. Finalmente, para que se visualice la ventana se realizará con el método `start` que visualizará una ventana con la configuración indicada anteriormente.

En el caso de querer añadir componentes como una barra de progreso, un botón o derivados, es tan sencillo como crear una sección y emplear el método `addSection`, el cual recibe por parámetro la sección a añadir, el nombre de la sección y la posición de este en la ventana.

¿Cómo funciona la sección?

Las secciones son instancias de la clase `Section` que permiten formar los diferentes componentes de la ventana.

El funcionamiento de una sección es muy sencilla, se trata de instanciar un objeto de la clase `Section` y llamar a algún método de esta clase, pasando los parámetros adecuados, y finalmente añadir la sección a la ventana.

Otros

Adicionalmente, se dispone de una clase llamada `DirectionAndPosition`, que constituye las posibles orientaciones y direcciones que una sección puede tener.

2.1.2. Implementación

Para el desarrollo de esta librería se ha centrado en la simplicidad hacia el usuario final y la eficiencia de código mediante funciones sencillas de emplear y optimizadas para asegurar un mayor rendimiento de la interfaz de usuario. Este paquete se divide en dos principales partes:

Window: Consiste en un conjunto de funciones para la creación y configuración de la ventana.

Section: Consiste en un conjunto de funciones base para la creación de secciones en la ventana.

La implementación de la `Window` consiste en diversas partes: gestión de teclas, configuración, creación y actualización de la ventana y creación de las secciones.

La gestión de las teclas se realiza mediante una clase llamada `KeyActionManager`, que implementa la interfaz `KeyListener`. Esta clase permite gestionar los eventos de teclado de la ventana y se utiliza para la depuración y el desarrollo del programa.

En cuanto, la configuración, creación y actualización de la ventana y la creación de las secciones, se han desarrollado una serie de métodos que envuelven a un conjunto de funciones propias de java swing. Por tanto, con este conjunto de métodos y funciones se obtiene la clase `Window`.

A continuación se explicarán los principales métodos de `Better swing`:

`initConfig` es un método que permite cargar la configuración de la vista y crea el marco de la vista, incluyendo apariencia, posición, tamaño, icono, color de fondo y manejo de eventos de teclado.

`start` permite la visualización de la ventana, haciendo que la ventana sea visible para el usuario.

`stop` actua como wrapper de `JFrame::dispose`.

`addSection` permite añadir una sección a un objeto `Window` indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`updateSection` permite actualizar una sección indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`deleteComponent` permite borrar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintComponent` permite repintar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintAllComponents` repinta todos los componentes de la ventana.

La implementación de la `Section` consiste en diversas partes, las cuales son las siguientes: Los métodos que permiten crear las secciones ya configuradas y las clases en las que se basan los métodos anteriores.

La propia librería contiene muchas más clases y métodos que si el lector desea explorar puede acceder a su documentación a través del código fuente proporcionado en la entrega de la práctica.

2.1.3. *Manual de uso*

En este apartado se describe como emplear la librería para su correcto funcionamiento.

Para emplear la librería es tan sencillo como crear una instancia de la clase `Window`, llamar al método `initConfig` indicando el fichero de configuración de la ventana, si se desea, en caso contrario se deberá pasar un `null` y se emplearán la configuración por defecto, y finalmente llamar a la función `start` cuando se desee inicializar la ventana.

Es importante inicializar la configuración antes de ejecutar la función `start`, ya que en caso contrario se producirá una excepción. A continuación se muestra un sencillo ejemplo:

```
Window view = new Window();
view.initConfig("config.json");
view.start();
```

Como se observa, la librería permite cargar la configuración e iniciar la ventana independientemente, permitiendo una mayor flexibilidad.

Además, en el caso de querer reiniciar la configuración, cambiar la visibilidad de la ventana o guardar el contenido de esta sin tener que volver a compilar el código, se puede realizar a través de unos atajos de teclado, que son los siguientes:

Q: Cerrar programa.
R: Reiniciar configuración.
V: Cambiar visibilidad.
G: Guardar contenido.

Por ejemplo, al cambiar la configuración y reiniciar la ventana se aplicarán los cambios automáticamente sin compilar de nuevo el código, es decir, se permite el conocido “Hot Reloading”.

En el caso de querer crear una sección y añadirla, se realizaría de la siguiente forma:

```
Window view = new Window();
Section section = new Section();
// Los datos pueden ser de longitudes irregulares
long[][] data = { ... };
Color chartColors[] = { Color.RED, Color.BLACK };
String chartColumnLabels[] = { "Linea 1",
                               "Linea 2" };
section.createLineChart(labels,
                        data,
                        chartColors,
                        chartColumnLabels,
                        "Ejemplo Lineas");
view.addSection(section,
                DirectionAndPosition.POSITION_TOP,
                "Chart");
```

En el ejemplo anterior se crearía una gráfica de líneas con dos líneas, una de color rojo y otra negra, con los nombres “Linea 1” y “Linea 2”, con el título “Ejemplos Lineas” y con los puntos del array data.

Finalmente, comentar que hay ilimitadas posibilidades de creación y personalización de componentes con los ya configurados y las posibilidades de crear libremente los propios componentes.

3. CONCEPTOS MVC

El Modelo Vista Controlador (MVC) es un patrón arquitectónico de software que divide una aplicación en tres elementos interconectados, separando la representación interna de la información, como se muestra al usuario y donde se hacen cómputos con esa información respectivamente.

Para esta práctica, se ha decidido modificar parcialmente este patrón al implementar la comunicación entre los módulos mediante un sistema de peticiones. Por ende, encontramos 4 módulos en nuestro MVC: Modelo, Vista, Controlador y Hub. Este último se encargará de gestionar toda la comunicación.

Esta decisión de modificación del MVC ha sido respaldada por la facilidad que obtenemos de escalar el patrón, permitiendo crear tantos módulos como queramos. Esto permitiría que, si en un futuro se quisiera añadir otro controlador en otro lenguaje de programación o encapsular controladores por funcionalidad, el único cambio a efectuar ocurriría en el hub.

A continuación se explicarán cada uno de los módulos y como han sido adaptados y empleados en la práctica.

3.1. Implementación

El Modelo vista controlador (MVC), en esta práctica, ha sido implementado como si fueran aplicaciones diferentes, es decir, cada parte del MVC es independiente de otro elemento de este ya que todos las partes se comunican a trvés de un hub. Esta implementación, trata de imitar una serie de microservicios que se comumican entre ellos a través de una API (application programming interface).

En cuanto a la implementación interna, se trata de una serie de servicios (modelo, vista y controlador) que se conectan a un servidor (hub), donde, entre ellos, se comunican a través de la red local del dispositivo.

¿Cómo hemos conseguido implementar esta idea en Java?

Para implementar el concepto de servicios y servidor se ha realizado a través de los sockets de Java, el problema, de únicamente emplear los sockets, es que los procesos se bloquean cuando envían una petición al servidor hasta que reciben una respuesta, por ello, hemos decidido crear una comunicación asíncrona no bloqueante, es decir, los servicios y el servidor no se pueden bloquear. El concepto empleado es similar al que usa el entorno en tiempo de ejecución multiplataforma NodeJs. Básicamente, un servicio realizará una petición y la respuesta que obtendrá es una promesa diciendo que le llegará una respuesta con los datos que desea, en el caso de que desee obtener información de vuelta, donde le informará de que en algún momento será notificado con los datos solicitados en la petición realizada. Con este planteamiento, se consigue que los procesos, como se ha mencionado anteriormente, no se bloquen y puedan seguir con su ejecución. En el caso, de que se desee esperar a la información, se puede realizar uso de una función similar a un "await", en la cual, se encargará de bloquear el proceso hasta que la petición tenga los datos esperados.

La siguiente imagen muestra un diagrama de cómo está implemtentado el sistema de comunicación entre los elementos:

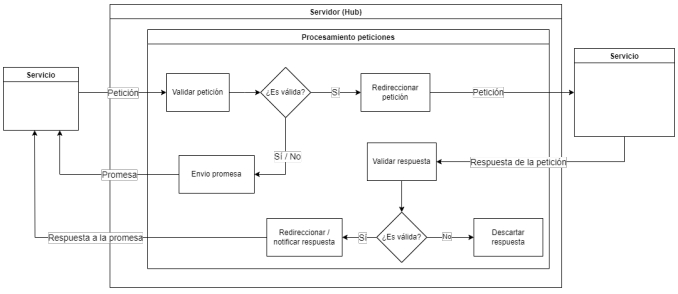

Figura 1. Comunicación entre elementos MVC y hub.

Además, de la implementación comentada anteriormente, los "endpoints"/rutas del servidor se definen a través de un archivo de configuración JSON, el cual, mapea cada petición con los servicios, al igual que sus respuestas pertinentes. Esto se puede ver en el siguiente ejemplo:

```
{
  "requestMap": [
    {
      "code": "SEND_GEOPOINTS",
      "services": ["MODEL", "CONTROLLER"]
    }
  ],
  "responseMap": [
    {
      "code": "LOAD_MAP",
      "services": ["VIEW"]
```

```
    },
  ],
}
```

3.2. Hub

El flujo de datos empleado en este patrón de diseño ha sido inspirado en cómo se comunican los diferentes elementos en un servidor. Es decir, cualquier módulo del MVC deberá realizar una petición al servidor (Hub) y este se encargará de solventar y dar servicio a dicha petición. Se puede ver con mayor claridad cómo funciona y su implementación en la siguiente figura:

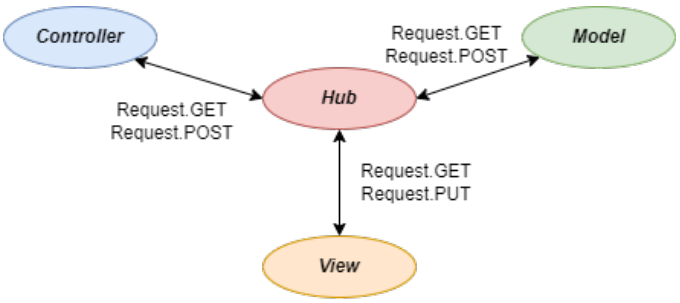

Figura 2. Patrón MVC usado en la práctica.

El Servidor/Hub se encarga de administrar los diferentes tipos de peticiones que llegan de los diferentes componentes. En la figura 2 se observan los diferentes tipos de peticiones posibles que se pueden realizar, en concreto: GET, POST y PUT. En un entorno web, GET sirve para obtener datos, POST para enviar y PUT para actualizar. En el programa los diferentes tipos de peticiones se han traducido en:

GET Realiza un retorno del dato solicitado.
POST Añade un el dato enviado.
PUT Realiza un set del dato enviado.

Para permitir esta implementación, se ha creado la interfaz `Service` con el método `notifyRequest` el cual deberá estar implementado en todos los módulos del MVC, ya que será el responsable de gestionar las peticiones y asegurar que cada uno de ellos puede realizar sus tareas designadas de manera eficiente y precisa.

3.3. Modelo

El modelo es la representación de los datos que maneja el software. Contiene los mecanismos y la lógica necesaria para acceder a la información y para actualizar el estado del modelo.

Para este proyecto, y con el objetivo de ser lo más fiel a un caso realista aplicable a una aplicación real, se ha decidido crear una `sqlite` donde se guardarán todos los datos. Así pues, nuestro modelo presenta dos elementos:

3.3.1. Base de datos

La base de datos consta del conjunto de los tableros con sus respectivas heurísticas y estadísticas; formalmente definidas como "Solucion". Técnicamente los atributos son:

- **Board**: El tablero al que se le ha aplicado la solución. A efectos prácticos, es un "wrapper" de un array bidimensional de "Integers" con un conjunto de métodos que permiten mutar su estado.
- **Heuristic**: La heurística utilizada en la ejecución. A efectos prácticos, es un enumerado con un método `apply` que retorna un "Integer" con el valor resultante de la heurística.
- **Movements**: El conjunto de movimientos necesarios para devolver el tablero a su estado original.
- **Stats**: Una estructura de datos que contiene un conjunto de métricas a tomar durante su ejecución. Técnicamente:
  - **statesVisited**: La cantidad de estados visitados.
  - **totalStates**: La cantidad total de posibles estados de un tablero.
  - **memoRef**: La cantidad de referencias que se han hecho a la memoización.
  - **memoHit**: La cantidad de referencias que han devuelto un valor no nulo.
  - **pruneCount**: La cantidad de nodos prunados.
  - **timeSpent**: El tiempo total de la ejecución.

3.3.2. Interfaz

Ya que el modelo es realmente la base de datos, es necesaria una interfaz que nos permita leer, modificar o escribir dentro de la base de datos. Para ello, la propia clase `Model.java` junto a `DBApi.java` nos permite ejecutar un conjunto de funciones que aplican queries a la base de datos. La mayoría de estos métodos siguen el siguiente patrón:

- Conectar a la base de datos.
- Ejecutar el query.
- Transformar el resultado a un objeto de java (como puede ser un array).
- Devolver el resultado si no hay errores.

Aunque el diseño se acerca a la posible implementación en una aplicación real, la implicación de usar queries a una base de datos nos proporciona tanto beneficios como perjuicios.

Beneficios

- Se delega la búsqueda y filtrado de datos a un lenguaje especialmente diseñado para ello.
- Los datos más usados se guardan en la caché, mientras que los demás se pueden guardar en disco.
- Los datos se mantienen entre ejecuciones, por lo que se puede aplicar un posterior estudio de los resultados en otros entornos y lenguajes.

Perjuicios

- Se genera un overhead general en la aplicación proporcional a cómo esté programada la librería usada, ya que la conexión se debe abrir y cerrar en cada query que se ejecuta, además de la necesidad de transformar los resultados a un objeto del lenguaje usado.

Aun así, tanto por interés académico como para crear ejemplos lo más cercanos a una aplicación real, se ha decidido seguir este acercamiento ante el problema de la gestión de un modelo en una aplicación.

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Service` y su método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4. Vista

La vista contiene los componentes para representar la interfaz del usuario (IU) del programa y las herramientas con las cuales el usuario puede interactuar con los datos de la aplicación. Adicionalmente, la vista se encarga de recibir e interpretar adecuadamente los datos obtenidos del modelo. Cabe mencionar, que al igual que el resto de componentes del MVC, la clase View implementa la interfaz “Service”, la cual permite la comunicación con el resto de elementos.

`loadContent` es el encargado de cargar el contenido inicial en la ventana. Para esta práctica, carga los siguientes elementos semánticos:

`menu`, función que crea y configura el menú de utilidades de la ventana principal. En esta, se incluyen las siguientes opciones: Abrir ventanas de estadísticas, abrir el manual de usuario, cargar la base de datos y salir del programa.

`body`, función que crea, configura y actualiza la visualización del puzzle. Se trata de una serie de botones que permiten con un texto o imagen que representan la forma del puzzle de manera visual y agradable.

`sidebar`, función que crea y configura la barra de opciones de la derecha de la interfaz principal. Esta permite al usuario interactuar y configurar el entorno de ejecución del algoritmo, para ello, se permite: seleccionar la imagen del puzzle, el tamaño y la heurística.

`footer`, función que crea y configura los botones para iniciar el algoritmo, entre ellos, barajar el puzzle actual, seleccionar una semilla para barajar el puzzle y un botón para iniciar el algoritmo.

Adicionalmente, a la vista principal, esta permite desplegar una serie de ventanas extra.

Una ventana muestra, a tiempo real, el uso y consumo de la memoria de la Java virtual machine (JVM) (3.4.1), la otra ventana muestra las estadísticas de la ejecución de los algoritmos además de su comparación (3.4.2) y un manual de usuario (4.1).

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Service` y su método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4.1. Estadísticas JVM

Este apartado de la vista principal, es el encargado de enseñar a tiempo real las estadísticas de la máquina virtual de java. Concretamente, se actualiza cada 0.5 s a partir de los datos obtenidos de la clase de java "Runtime" y muestra la memoria libre, la memoria total y el uso de esta en una gráfica de líneas, donde el eje x es instante en el tiempo que se han obtenido los datos y el eje y su valor. Todos los datos de la memoria obtenidos están en MB.

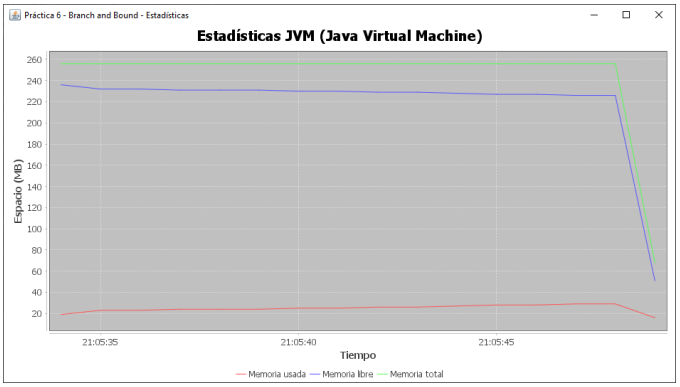

Figura 3. Interfaz estadística JVM

3.4.2. Estadísticas de los Algoritmos

Este apartado de la vista principal, es el encargado de enseñar las estadísticas de la ejecución del algoritmo de "Branch and Bound". Estas estadísticas incluyen una gráfica con el tiempo de ejecución de cada ejecución del algoritmo. Los resultados (tiempos de ejecución) son representados en un gráfico de barras, donde el eje x representa el número de ejecuciones del algoritmo y el eje y el tiempo en milisegundos que ha tardado. Dos gráficos circulares comparando los ratios de las veces que se ha referenciado, podado el árbol y cuantos se han visitado. Y finalmente, un gráfico de barras con la cantidad de movimientos realizados para encontrar la solución por heurística.

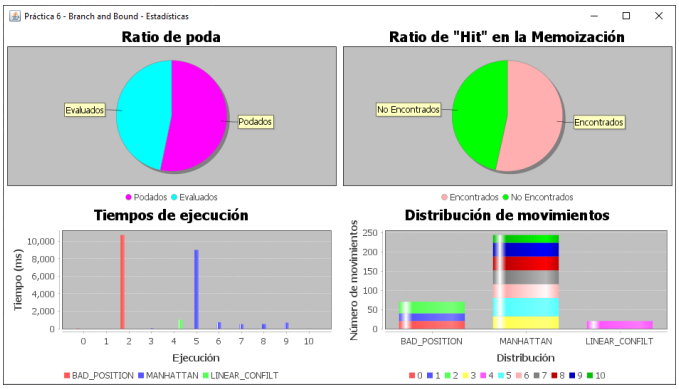

Figura 4. Interfaz estadísticas algortimos

Como se ha podido ver anteriormente en la imagen (4), se puede apreciar los diferentes datos obtenidos tras una serie de ejecuciones del algoritmo.

3.5. Controlador

El controlador en un MVC es el responsable de recibir y procesar la entrada del usuario y actualizar el modelo. En esta práctica, dicha responsabilidad es la de solucionar el problema del "Puzzle 15" para tableros de tamaño genéricos mediante "Branch & Bound"

3.5.1. Diseño

Esencialmente, el controlador solo presenta un método: Solucionar un tablero dada una heurística. El algoritmo desarrollado tiene como objetivo encontrar la solución óptima explorando el espacio de posibles movimientos dentro de un tablero. Para ello, y por motivos académicos, se decidió tomar una perspectiva no recursiva, optando por la creación de una cola de prioridad. Esto no solo nos permite simular el comportamiento de un algoritmo recursivo, ya que podemos explorar los nodos más óptimos en cualquier momento, evitando la exploración de nodos menos "apropiados" en niveles más profundos.

Adicionalmente, se han incorporado nociones de programación dinámica al observar que muchos de los estados de la tabla se repetirán debido a la naturaleza del espacio de movimientos. En el apartado 3.5.2 se puede observar el beneficio obtenido mediante un conjunto de casos de prueba.

En pseudocódigo, el algoritmo podría tener la siguiente forma

```
1  def solve(Board, Heuristic):
2      pq = PriorityQueue(e -> cost(e, Heuristic))
3      pq.add((Board, []))
4
5      memo = HashMap()
6      memo.add(Board, cost(Board))
7
8      lower_bound = INT.MAX
9      best_solution = None
10
11     while(pq.has_values()):
12         best_board, movements = pq.take()
13         cost = cost(best_board)
14
15         if(cost > lower_bound):
16             continue
17
18         if (best_board.is_solved()):
19             lower_bound = cost
20             best_solution = (best_board, movements)
21             continue
22
23         for(movement in possible_movements):
24             moved_board = best_board.move(movement)
25
26             if (moved_board in memo):
27                 continue
28
29             memo.put(moved_board, cost(moved_board,
               ↪  Heuristic))
30             pq.add((moved_board, movements +
               ↪  movement))
31
32     return best_solution
```

Como se puede observar, de la línea 15 a la 21 se gestiona todo el "Branch & Bound" mientras que las líneas 26 y 27 gestionan la memoización.

Con respecto a la complejidad temporal estricta, obtenemos que presenta $O(4^{N^2})$, ya que, en el peor caso posible y suponiendo que siempre se puede mover en las cuatro direcciones cartesianas, el algoritmo deberá explorar todos los posibles estados hasta encontrar la solución; necesitando iterar sobre N^2 elementos y por cada uno de ellos efectuar 4 movimientos.

Con respecto a la complejidad espacial estricta presentamos un conjunto de estructuras que guardan información extra:

- **Priority Queue**: Guarda los todos los nodos a explorar, tomando $O(\mid E \mid)$ donde $\mid E \mid$ es el número de elementos de la cola de prioridad
- **Memoization**: Guarda todos los estados visitados, tomando $O(\mid V \mid)$ donde $\mid V \mid$ es la cantidad de estados guardados
- **currentSol**: Necesita tanto espacio como movimientos tenga la solución. Debido a que esta cantidad es indefinida al depender de la heurística usada, lo tomaremos como $O(\mid M \mid)$ donde $\mid M \mid$ es la cantidad de elementos

Así pues, el coste espacial del algoritmo sería de $O(\mid E \mid + \mid V \mid + \mid M \mid)$.

3.5.2. Estudios

3.5.2.1. Memoización: Para poder apreciar el beneficio explícito de añadir memoización encima de un algoritmo de poda, debemos acuñarnos en las métricas obtenidas para una ejecución cualquiera. Dado una tabla 5x5 mezclándola 150 obtenemos los siguientes resultados:

- **memo refs** : 721769
- **memo hits** : 239604 (33.19677 %)
- **prunations** : 266513
- **visited states**: 482166

Si tomamos en cuenta solo las prunaciones que se hacen, de los 482166 solo evaluamos 215653, siendo esencialmente la mitad. Sin embargo, de ese 50 % aproximado que evaluamos, al estar memoizando los estados de la tabla, se hace "hit" a la memoización un 33 % de veces de los nodos que evaluamos. Esto nos reduce de 50 % a aproximadamente 33 %. Esto implica que de los 482166 nodos totales que hemos visitado a lo largo de la computación realmente solo se ejecutan computaciones a 159114. Así pues, al introducir memoización en el algoritmo, reducimos la cantidad de nodos a evaluar sustancialmente.

3.5.2.2. Heuristicas: Al estar escogiendo que nodo evaluar primero mediante una heurística cualquiera, permitimos por diseño la aplicación de una poco óptima para el problema en cuestión. Así pues, y conociendo el dominio del problema, podemos aplicar heurísticas que se adapten más fuertemente a nuestro problema. Una clara heurística aplicar sería la distancia de Manhattan, ya que el dominio de movimientos se rige mediante este tipo de movimientos. Así pues, y para el mismo estado inicial, podemos comparar los beneficios de utilizar heurísticas específicas al problema. Se han ejecutado 5 tableros de diferentes tamaños y cantidad de movimientos para mezclarlos y estos son los resultados:

Bad Position	**Manhattan**
▪ 7ms	▪ 0ms
▪ 4ms	▪ 1ms
▪ 946ms	▪ 87ms
▪ DNF	▪ 179ms
▪ DNF	▪ 304ms

Como se puede apreciar, los datos posicionan a la heurística de Manhattan como la clara vencedora. Esto nos enseña que, aunque un algoritmo esté perfectamente diseñado, el tener una función de prioridad menos específica para el problema en cuestión puede afectar significativamente al rendimiento general de la aplicación.

4. MANUAL DE USUARIO

Para el correcto uso de la aplicación, el conjunto de acciones que se pueden realizar en dicho programa serán definidas a continuación. También, la distribución de las secciones de la interfaz junto a su explicación y funcionalidad.

Cuando se ejecute el programa por primera vez en la pantalla se debe mostrar la siguiente interfaz de usuario:

4.1. Menu

El menú de la aplicación consiste en la barra que se sitúa debajo del marco superior de la ventana. En esta se puede encontrar un conjunto de opciones para que el usuario pueda interactuar con la aplicación, modificar y analizar el comportamiento de esta. En concreto, se encuentran las opciones de “Opciones”, “Datos” y “Ayuda”, respectivamente.

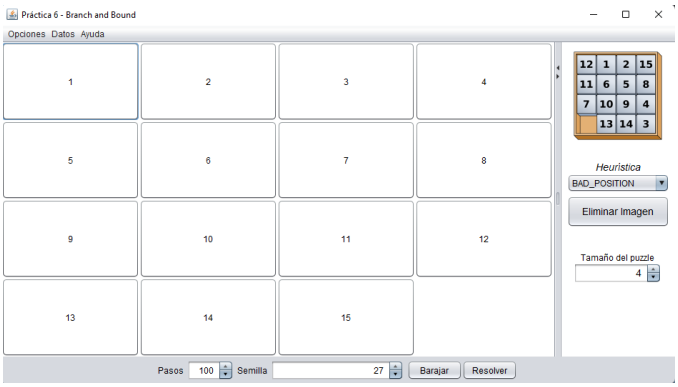

Figura 5. Interfaz de usuario.

En la primera se sitúan las acciones para salir del programa y para iniciar las ventanas de estadísticas explicadas anteriormente en los apartados 3.4.1 y 3.4.2.

En la segunda opción llamada “Datos” encontramos en primer lugar, la opción para crear una base de datos y cargar los datos. Por otra parte, tenemos la opción de cargar una imagen directamente.

En la tercera opción se encuentra un menú desplegable con un manual de usuario con la explicación del funcionamiento de la aplicación.

4.2. Main

El “Main” es el bloque principal de la vista, donde se representará el problema del Puzzle 15. El usuario no podrá interactuar directamente con las casillas para moverlas, sino que, tendrá diferentes opciones para poder resolver el problema. Entre las opciones se encuentran: la heurística, el tamaño del puzzle, los pasos para resolverlo, la semilla, un botón para mezclar las casillas y por último, otro botón para resolver el problema.

4.3. Sidebar

El "Sidebar" contiene un conjunto de opciones para modificar los datos y el modo de ejecución de la aplicación. A continuación, se explicará cada una de estas opciones y su función.

En primer lugar, se encuentra la opción para seleccionar la "Heurística". En esta, se puede escoger el método de resolución del algoritmo a aplicar sobre el problema. Se han definido las siguientes heurísticas:

- BAD POSITION
- MANHATTAN
- LINEAR CONFLICT

Debajo de este, se encuentra un botón para eliminar la imagen actual y volver a la primera pantalla.

En tercer lugar se encuentra la opción para seleccionar el tamaño del puzzle, el tamaño mínimo será de 2x2 y el tamaño máximo de 50x50.

4.4. Footer

En la sección "Footer", se hallan varias opciones para interactuar con la interfaz y también con el algoritmo a ejecutar.

En primer lugar tenemos una opción para seleccionar el número de veces que se va a realizar el proceso de barajado.

En segundo lugar la semilla de aleatoriedad para generar el número aleatorio.

En tercer lugar, encontramos el botón "Barajar", este botón realiza la mezcla de las casillas, cambiando las posiciones de unas por otras. Se puede clicar las veces que el usuario desee.

Por último, el botón "Resolver" que ejecutará el algoritmo con los parámetros que le hayamos indicado en la UI.

4.5. Ejemplo ejecución

Al iniciar la aplicación se mostrará una interfaz como la expuesta en la imagen 5. Para poder iniciar la ejecución del algoritmo, el usuario tendrá que mezclar las casillas, y pulsar el botón de resolver. Además, tiene la opción de cambiar los pasos o la semilla de aleatoriedad. Por otra parte, el usuario podrá seleccionar la heurística para la resolución del problema y añadir imágenes para hacerlo visual. A continuación, la siguiente imagen, muestra un ejemplo de la interfaz tras la ejecución:

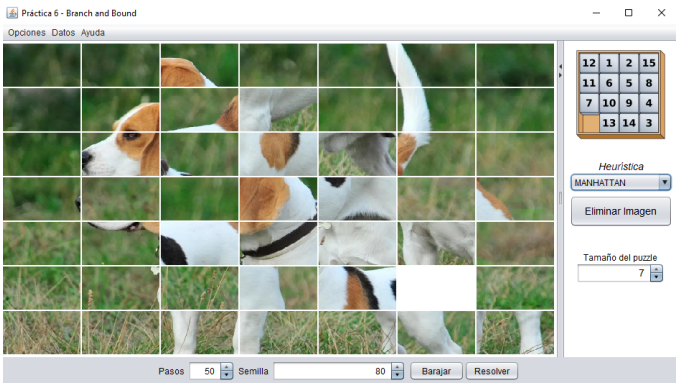

Figura 6. Interfaz de usuario en ejecución.

Tras ejecutar el algoritmo, se mostrará al usuario paso por paso como se resuelve el mismo.

A partir de aquí, el usuario puede salir del programa con el botón del menú, cargar una base de datos, cargar una imagen, cambiar la heurística, el tamaño del puzzle o las imágenes y ver estadísticas de la ejecución, ver apartados 3.4.1 y 3.4.2.

5. Conclusión

Teniendo en cuenta el desarrollo expuesto, la aplicación permite generar un punto de referencia de un conjunto de algoritmos de diferente complejidad asintótica mediante una representación gráfica e interactiva de la media de los tiempos de ejecución a través de unos parámetros de entrada definidos por el usuario. Adicionalmente, se ha implementado mediante una versión modificada del patrón de diseño de software **M**odelo **V**ista **C**ontrolador (MVC)

añadiendo un cuatro módulos que permite la comunicación entre los diferentes elementos de este mediante peticiones. Esta modificación ha sido fuertemente inspirada en el diseño de un servidor, lo que permite gran flexibilidad y escalabilidad de desarrollo, al poder interceptar y gestionar las peticiones a gusto del desarrollador.

Además, se ha hecho énfasis en el uso de las librerías creadas por los miembros del grupo para facilitar el trabajo y la reutilización de código en futuras prácticas. Reiterar que el principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo "overhead" al rendimiento del programa.

Concluir que, con respecto al ámbito académico, este proyecto nos ha permitido consolidar el concepto de patrón de diseño MVC gracias a un primer proceso de investigación y discusión del diseño a implementar entre los integrantes del grupo.

6. Distribución del trabajo realizado

El trabajo realizado por cada miembro de la práctica ha sido el siguiente:

- Alejandro Rodríguez:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de la UI del proyecto
  - Desarrollador de la guía de usuario
  - Desarrollador de la plantilla base del proyecto
- Rubén Palmer:
  - Documentación
  - Diseño del proyecto
  - Principal desarrollador del "backend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "frontend"
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto
  - Desarrollador de la plantilla de la documentación
- Sergi Mayol:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de la librería `Better Swing`
  - Principal desarrollador del "frontend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "backend" de la aplicación
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto

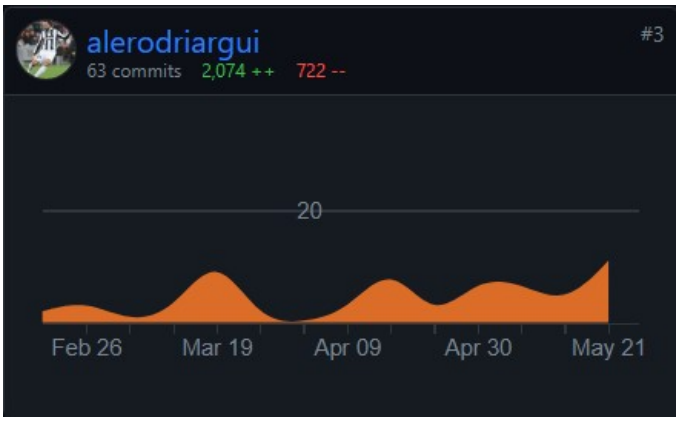

Figura 7. Contribución en el proyecto de Alejandro Rodríguez

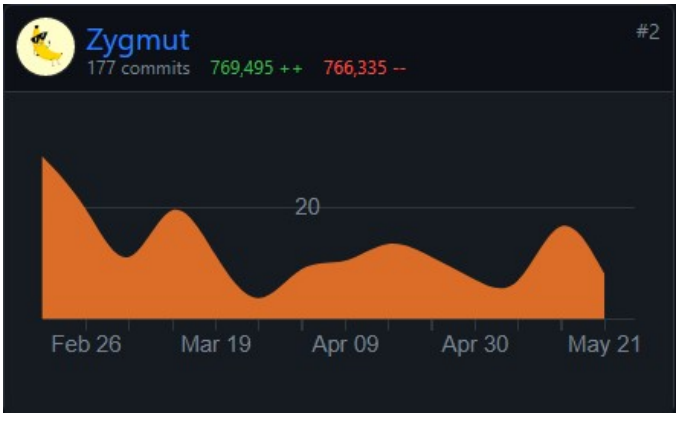

Figura 8. Contribución en el proyecto de Rubén Palmer

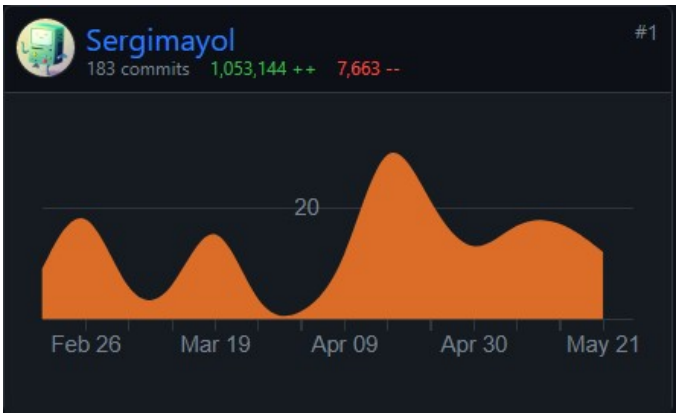

Figura 9. Contribución en el proyecto de Sergi Mayol

Reconocimientos

- Se agradece la colaboración del Dr. Miquel Mascaró Portells en la resolución de dudas y supervisión del proyecto.

Referencias

[1] Better Swing, *An easy way to develop java GUI apps*, V0.0.3. Ver documentación completa aquí.

[2] Graphical engine, *How to develop a graphical engine from scratch*. Ver enlace aquí.

[3] Google json (Gson), *Java serialization/deserialization library to convert Java Objects into JSON and back*. Ver enlace aquí.

[4] Node.js Architecture, *Architecture of a single thread cross-platform, open-source server environment*. Ver enlace aquí.

[5] The SQLite database, a serverless database. *Quick start and documentation*. Ver enlace aquí.

[6] Open source Java chart library. *An easy way to create and visualize charts in Java*. Ver enlace aquí.

[7] A way to define different behavior for a method with the same name. Ver enlace aquí

[8] What is the puzzle 15 game? How does it work?. Ver enlace aquí

[9] Branch and bound concept and algorithm. Ver enlace aquí