Capítulo 6

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

Julian Wallis Medina
Universitat de les Illes Balears
Palma, España
julian.wallis1@estudiant.uib.cat

La ramificación y poda es una técnica de programación muy útil para algoritmos con solución basada en árboles. En este documento se propone una solución al problema del puzle N-1 basado en dicha técnica utilizando un conjunto de distintas heurísticas y comparando el rendimiento de estas.

Keywords—MVC, rendimiento, algoritmo, ramificación y poda, puzzle NxN.

I. INTRODUCCIÓN

La ramificación y poda es un paradigma de diseño algorítmico generalmente utilizado para la resolución de problemas de optimización combinatoria. Guarda una gran similitud con otro paradigma ya estudiado, el backtracking. Ambos comparten el concepto de árbol de soluciones, el cual se deberá de ir recorriendo hasta llegar a una solución válida. La principal diferencia radica en la manera en la que se recorre dicho árbol (más adelante se comentarán otras diferencias).

Este paradigma consiste en una enumeración sistemática de las soluciones candidatas. El algoritmo explora las distintas "ramas", las cuales representan subconjuntos del conjunto solución. Antes de enumerar las soluciones candidatas de una rama, dicha rama se revisa contra los límites estimados superior e inferior de la solución óptima y se descarta (poda) si no puede producir una solución mejor que la mejor encontrada hasta ahora por el algoritmo.

Con este algoritmo se pretende dar solución a uno de los problemas típicos de la programación, el "Puzzle 8". Es el mítico juego en el que se nos da una cuadrícula 3x3 con una ficha por casilla enumerada del 1-8. Dejando así una posición vacía (o en blanco). Una vez mezclado el puzle el objetivo es llegar al estado inicial con todas las fichas en su orden. Para ello se deberá ir moviendo la ficha vacía, las operaciones permitidas son:

- Norte (Arriba)
- Sur (Abajo)
- Este (Izquierda)
- Oeste (Derecha)

En esta práctica en lugar de números de opta por una disposición sobre una imagen, en la que uno llega a la solución cuando la reconstruye.

Todo el desarrollo de la práctica se realizará siguiendo el patrón de diseño MVC (Modelo Vista Controlador). El cual permite que los proyectos sean mucho más escalables, además de tener un mejor control de errores. Todo esto a costa de una mayor complejidad de código.

II. CONTEXTO Y ENTORNO DE ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre varios IDE's (*Integrated Development Environment*).

- NetBeans
- IntelliJ
- VS Code

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su

versión de escritorio *Github Desktop* por su facilidad de uso mediante interfaz gráfica.

III. DESCRIPCIÓN DEL PROBLEMA

El "Puzzle 8" o hablando en términos generales "Puzzle N" es un rompecabezas de fichas deslizantes que se juega sobre una matriz nxn, dejando o una posición vacía o un recuadro en blanco (para permitir el desplazamiento del resto). En el juego original las fichas están numeradas. El objetivo es llegar al estado inicial (con las fichas ordenadas) una vez mezclado el rompecabezas.

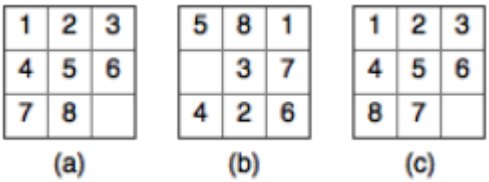

Ilustración 1(a) Estado objetivo del Puzzle 8, (b) Estado no objetivo y (c) Estado irresoluble

Como se puede observar en la *Ilustración1* otro de los problemas que se ha afrontado durante el desarrollo de la práctica es la de la generación de un estado a resolver permitido, es decir, que con las reglas definidas sobre el juego se es capaz de llegar a una solución.

En la imagen *c* se puede observar un claro ejemplo de estado no permitido. Ya que por muy bien que funcione el algoritmo, jamás podrá llegar al estado *a*. Esto es debido a una mala implementación de la función encargada de mezclar el puzzle. Una primera aproximación errónea sería la de generar una disposición completamente aleatoria, lo que muy probablemente desembocaría en un estado irresoluble. Para ello, se ha de mezclar el puzzle siguiendo los movimientos definidos, ya sobre estos se podrá aplicar un movimiento aleatorio de los disponibles.

A. *Complejidad del problema y uso de B&B*

Los casos posibles iniciales de un puzzle de nxn es de $n!$. Ahora bien, como bien se ha señalado antes, hay que descartar todos aquellos casos que son irresolubles. En el año 1879 Johnson & Story llegaron a la conclusión de que el número de estados iniciales posibles se reducía a la mitad. Quedando así la complejidad del problema en $\frac{n!}{2}$.

Aunque a la hora de resolver el problema, pueda parecer una gran ventaja el poder reducir el tamaño del problema a la mitad de los casos, no se debe de olvidar que en cuanto a coste asintótico se refiérela complejidad seguirá siendo de $n!$. Esto es debido a que el factorial es de orden superior a la división por un número entero.

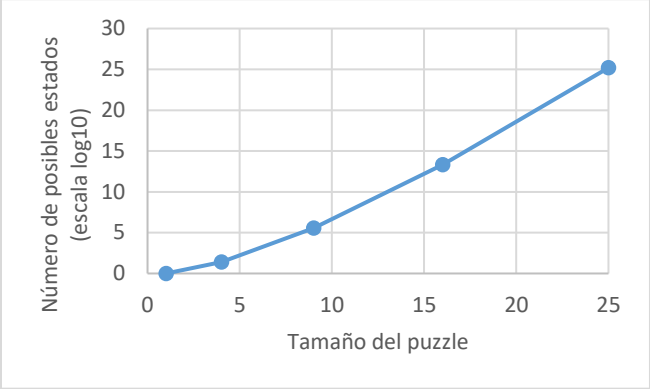

Gráfico 1

Como se puede observar en el *Gráfico 1* el número de estados posibles se dispara al poco tiempo. Hay que tener en cuenta que se han normalizado las y's ($\log_{10} n$) porque sino era impracticable cualquier tipo de representación gráfica.

De cara a la aplicación de distintos algoritmos para encontrar una solución, este problema es muy llamativo ya que, aunque las normas son sencillas y fáciles de implementar, el número de casos posibles es altísimo. Por lo que es un buen campo de pruebas para comparar las distintas heurísticas.

IV. SOLUCIÓN PROPUESTA

En este apartado se comentará a modo de carácter general el planteamiento seguido en la solución implementada. Los detalles técnicos serán comentados en apartados posteriores, especialmente en la implementación del modelo.

Se ha implementado una propia estructura de nodo ya que la propia de Java no suplía las necesidades para resolver este problema. Cada nodo representa un estado del tablero el cual se ha llegado con un movimiento determinado. Además, permite conocer tanto el coste asociado para llegar a dicho estado, así como de donde viene (nodo padre). También se ha utilizado una cola de prioridad en la que estarán ordenados los nodos de menor a mayor coste.

El proceso sería el siguiente: Se añade la raíz a la cola de prioridad y mientras esta no esté vacía se va iterando. Se saca el primer nodo de la cola y se analiza su coste, si es 0 significará que se ha llegado a la solución y el programa terminará. En caso de que no sea así se generan todos los posibles movimientos (serán máximo 4) y se crea un nuevo nodo con cada uno de los movimientos. Si existe un nodo con las mismas características que el creado dentro de la cola, no se

introduce. Este es el caso en el que se ha llegado a un mismo estado por otro camino, pero como el camino ha llevado más estados intermedios, su coste será peor (por lo que no se introduce). En caso contrario, se calcula el coste del nodo con la heurística correspondiente y se añade a la cola.

V. PATRÓN MVC

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[1] en el que se divide la lógica del programa de su representación gráfica, además se hace uso de un controlador para los eventos y comunicaciones entre las distintas partes.

Este patrón de arquitectura se basa en las ideas de reutilización de código y la separación de conceptos distintos, estas características pretenden facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Para ello se proponen la construcción de tres componentes:

- Modelo:
  Es donde se almacena la información necesaria para la ejecución de la aplicación. Gestiona el acceso a dicha información mediante peticiones a través del controlador. También contiene los procedimientos lógicos que hagan uso de esa información.

- Vista:
  Contiene el código que muestra la aplicación, es decir, que va a producir la visualización tanto de la interfaz del usuario como de los resultados

- Controlador:
  Responde a eventos usualmente generados por el usuario y los convierte en comandos para la vista o modelo. Se podría decir que hace la función de intermediario entre la vista y el modelo encargándose de la lógica del programa.

Como se ha mencionado anteriormente se ha decidido utilizar este patrón debido a la facilidad que aporta al programar la separación de conceptos. Además, se obtiene una capa más de abstracción ya que es posible utilizar diferentes vistas con un mismo modelo.

Las ventajas principales del MVC son: Escalabilidad, facilidad de tratamiento de errores y reutilización de componentes. Existen otras ventajas, pero esta arquitectura se aprovecha en mayor medida en aplicaciones web, el cual no es este caso.

La principal desventaja que existe del patrón MVC es la complejidad que añade a la programación. Ya que, para el mismo problema, hay que modificar el acercamiento para que quepa dentro de esta arquitectura. Lo que implica una mayor complejidad.

Hoy en día, es muy común el uso de la programación orientada a objetos (POO) como paradigma principal, por ello cada componente del patrón MVC suele implementarse como una clase independiente. A continuación, se explica que proceso se ha seguido y como se ha implementado cada parte.

A. Implementación del Modelo

En el Modelo se definen las funciones que implementan nuestro algoritmo. Tenemos una clase Nodo, que representará un estado del tablero, en forma de matriz, y tendrá un puntero a su estado anterior (Nodo Padre).

El algoritmo principal se implementa en la función *solve*, pero la llamada desde el controlador se hace a la función *getSolution()*, debido a que la función *solve* nos devuelve el nodo solución, y en getSolution obtenemos el camino del árbol que llega a dicha solución, y hacemos el *reverse* para que la Vista pueda animarlo.

En la función *solve* se implementa la solución propuesta: tenemos una estructura de datos de tipo Cola de Prioridad, que contendrá los Nodos (estados) que hayamos descubierto del árbol de movimientos. Esta cola se ordena mediante el coste heurístico de cada nodo más su profundidad. Es decir, cuánto más profundo y más coste heurístico, peor.

El algoritmo va cogiendo el mejor Nodo de la Cola, y a partir de ese, genera los posibles movimientos desde ese estado (que serán, como máximo, 4 movimientos). Para cada movimiento, creamos un Nodo nuevo, con su movimiento, matriz de estado y Nodo padre. Antes de introducirlos en la Cola de Prioridad, primero comprobamos que no esté ya dentro. Por eso, la clase nodo tiene implementado una función *equals*, que devuelve que son iguales si tienen exactamente la misma matriz estado.
Si el Nodo ya estaba en la cola, no lo volvemos a introducir. En caso contrario, calculamos su coste heurístico con la Heurística seleccionada por el usuario, y introducimos el Nodo en la Cola.

[1] Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software.

El algoritmo parará cuando no haya encontrado solución (haya traversado todos los estados), o cuando el coste heurístico sea 0 (significa que hemos encontrado la solución), momento en que devolveremos el nodo elegido.

B. Implementación de la Vista

Para la implementación de la vista se ha creado la clase *View*.
La clase *View* es la ventana principal del programa que contiene tanto los componentes de la interfaz de usuario como toda la lógica que hay detrás de estos componentes.

La ventana principal contiene los siguientes elementos:

- **JButtons**:
  - Load Image: Permite que el usuario cambie la imagen del fondo
  - Shuffle: Mezcla el tablero automáticamente
  - Solve: Llama al algoritmo de resolución para que resuelva el puzzle
- **JComboBox**: Permite que el usuario elija la heurística del algoritmo
- **JSpinner**: Permite que el usuario elija el tamaño del tablero, está definido para que solo puede ser de tamaño 3, 4 y 5
- **JPanel**: Panel donde situaremos las casillas
- **JLabel[]:** Cada JLabel representa una casilla
- **JProgressBar**: Barra de progreso para dar feedback al usuario. Mientras el algoritmo está resolviendo el puzle estará en modo indeterminado, pero cuando encuentre una solución mostrará el porcentaje por el que va.

Hay definidos varios métodos, pero solo explicaremos con mayor detalle los más interesantes y que aportan valor a la documentación:

- *updateImgPanel():*
  Cada vez que se llama a este método se actualiza el panel de imágenes con la imagen actual (al inicio es "default.jpeg" pero se puede cambiar por cualquier imagen).
  Lo que hacemos es trocear la imagen dependiendo del tamaño del tablero y ponemos cada fragmento en su lugar, exceptuando el último fragmento que quedará nulo ya que necesitamos una casilla vacía.

- *swap(int movement):*
  Método que dada la dirección del movimiento intercambia la casilla vacía con la casilla en dicho movimiento.

- *swapTiles(MouseEvent evt):*
  El JPanel que representa el tablero tiene asociado un MouseListener para que cuando sea clique sobre una casilla adyacente a la vacía podamos intercambiarlas.
  Esto lo hacemos gracias a este método. Buscamos la posición del clic dentro del panel y sabiendo el tamaño del tablero y la posición de la casilla vacía podemos averiguar si la casilla clicada es adyacente a la vacía. Si lo es llamaremos al método swap con la dirección correspondiente.

- *shuffle():*
  Método que permite mezclar el tablero de manera automática.

  Un problema que presenta mezclar el tablero automáticamente es que si creamos un estado de manera totalmente aleatoria cabe la posibilidad de que generemos un estado no resoluble. Esto se debe a que hay estados que son posiciones no resolubles, es decir no existe ninguna posible solución.

  Una manera de solucionar este problema es generando un estado aleatorio y luego comprobar sí que es resoluble mediante un método de comprobación.

  Nosotros hemos adoptado otro enfoque. Si partimos de la posición inicial y hacemos M movimientos acabaremos con otra posición legal ya que si revirtiéramos todos los movimientos acabaríamos otra vez con la posición inicial. Al ser esto cierto no hace falta que partamos del estado inicial, si no que solamente basta que partamos de una posición legal.
  Así que nosotros empezamos desde la posición inicial y efectuaremos movimientos sin entrar nunca en un estado ilegal. De esta manera podremos asegurar que nuestro tablero siempre se podrá resolver.

  Por tanto, llevamos a cabo $10 + N^2$ movimientos aleatorios legales no repetitivos sobre el estado actual del tablero.

  Un movimiento legal consiste en efectuar un movimiento que cumpla las reglas y que el tablero quede en un estado legal. Básicamente nos aseguramos de no intercambiar piezas diagonalmente ni sacar la pieza vacía fuera del tablero (Por ejemplo, si está en el borde

superior y generamos el movimiento "UP" sería ilegal ya que no puede salir del tablero)

Un movimiento no repetitivo consiste en no llevar a cabo el mismo movimiento que el anterior. Hacemos esto para evitar revertir los movimientos anteriores.

El número de movimientos que generamos lo hemos decidido para que dependa del tamaño del tablero. Para el tablero de 3x3 generaríamos 19 movimientos aleatorios y para un 5x5 35

C. *Implementación del Controlador*

Para la implementación se ha creado la clase *Controller* que contiene todos los métodos necesarios para poder comunicar ambas partes del patrón MVC.

En primer lugar, se han definido dos atributos de la clase que contienen las instancias del modelo y la vista, estos se reciben directamente como parámetros a través del constructor. A continuación se ha definido el método *start()* que es al que debe llamar la aplicación principal para comenzar la ejecución de la aplicación. Este método únicamente añade los *listeners* [2] correspondientes al modelo y a la vista, posteriormente hace la vista visible al usuario.

Finalmente hay definidos dos métodos que son los que se han proporcionado como *listeners*. Estos métodos se ejecutarán cuando el modelo o la vista lancen el evento pertinente. A continuación, se explican en mayor detalle:

- *modelPropertyChange():*
  Este método como indica su nombre se ejecuta cuando el modelo cambie alguna de sus propiedades. En esta aplicación únicamente se ha tenido en cuenta la propiedad de *Update* que proporciona el tablero del nodo que se está explorando actualmente, esto se ha utilizado para ver el funcionamiento del algoritmo y poder hacer *"debug"*.

- *viewActionPerformed():*
  Este método como indica su nombre se ejecuta cuando la vista lance un evento. En esta aplicación únicamente se ha tenido en cuenta el evento que se produce al presionar el botón *"Solve"* ya que el resto de los botones de la vista se gestionan internamente en esta.

  Cuando este método recibe un evento nuevo se obtienen los datos de la vista y se transmiten al modelo para que realice los cálculos oportunos. Para que la vista no se quede congelada durante el proceso, se ha hecho uso del concepto de *Programación Concurrente* que se comenta a continuación.

1) *Programación Concurrente*

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte. Para este proyecto se ha decidido utilizar este concepto sobre el modelo para realizar los cálculos en segundo plano.

Como se ha comentado anteriormente en el método *viewActionPerformed()* se crea un nuevo hilo donde se ejecutaran los cálculos. Para ello se hace uso de la interfaz *Runnable* que nos permite crear un método *run()* que será el que ejecute el nuevo hilo. Dentro de este método se realizan las operaciones correspondientes, se obtienen los datos de la vista, estos después se pasan al modelo para que los compruebe y finalmente se muestran los resultados de nuevo en la vista.

VI. HEURÍSTICAS

A la hora de hacer la ramificación y poda se pueden usar gran variedad de heurísticas. Nosotros, con el fin de poder hacer distintos estudios de rendimiento, hemos implementado cinco heurísticas diferentes. Son:

A. *Poisiciones incorrectas*

Esta es la heurística más simple. Para calcular el coste de la heurística simplemente sumaremos el número de casillas que están en una posición incorrectas. Teniendo el peor estado un coste de N*N y el mejor un coste de 0 (siendo el mejor estado el estado resuelto).

B. *Distancia de Manhattan*

Esta heurística se basa en la distancia de Manhattan. La distancia de Manhattan de una casilla es el número de movimientos necesarios para llevar una casilla para llevarla a la posición correcta si pudiera

[2] Un *Event Listener* es un procedimiento o función en un programa que espera a que ocurra un evento. Ejemplos de un evento son el usuario haciendo clic o moviendo el ratón, presionando una tecla en el teclado.

moverse libremente en horizontal y diagonal. Por ejemplo, si tenemos el estado:

2 5 8
4 3 6
7 1 X

La casilla 1 tendría un coste de 1 + 2 = 3
Con esta heurística el peor estado con una N tendría un coste de 24.

C. *Distancia euclidiana*

Esta heurística también se basa en las distancias entre la posición actual y la correcta. La diferencia entre esta heurística con la de Manhattan es que en esta calcularemos la distancia euclidiana, es decir la distancia en "línea recta" entre la posición actual y la correcta.

D. *Columna Incorrecta, Fila Incorrecta*

Esta heurística se basa en la suma del número de casillas que están en la columna incorrecta y las que están en una fila incorrecta.
Con esta heurística el peor estado, que es ese donde todas las casillas están en una columna y fila incorrecta, tendría un coste de 2*N*N.
Ejemplo de un estado con coste máximo:

5 X 8
3 7 1
6 4 2

E. *Casillas adyacentes reversibles*

Esta heurística se basa en añadir una penalización a la distancia de Manhattan.
Por cada pareja de casillas adyacentes reversibles penalizaremos el coste con 2 movimientos (uno para cada casilla) Estas penalizaciones siempre vendrán en pareja.

Dos casillas son adyacentemente reversibles si son contiguas y cada una está en la posición de la otra.
Hemos decidido implementar esta penalización ya que las casillas adyacentemente reversibles son más costosas de tratar porque una tiene que moverse alrededor de la otra perturbando a las casillas del alrededor.

Ejemplo de un estado con parejas de casillas adyacentes reversibles:

2 1 6
5 4 3
8 7 X

Como vemos las parejas de casillas (1,2), (3,6), (4,5) y (7,8) son adyacentes y ambas están en la posición de la otra. Así que tendría un coste de 8. Este coste sumado a la suma de las distancias de Manhattan resultaría en un coste final para este estado de: 8 + 8 = 16.

VII. ESTUDIOS DE RENDIMIENTO

En este apartado se mostrarán los resultados de unas pruebas de rendimiento que se han realizado.
Estas pruebas han consistido en la ejecución del algoritmo de resolución, con las distintas heurísticas y tamaños (para 3x3 y 4x4).
Para N=3, se han ejecutado 70 soluciones para cada heurística, y se ha guardado el tiempo que ha tardado cada ejecución. Los resultados han sido los siguientes:

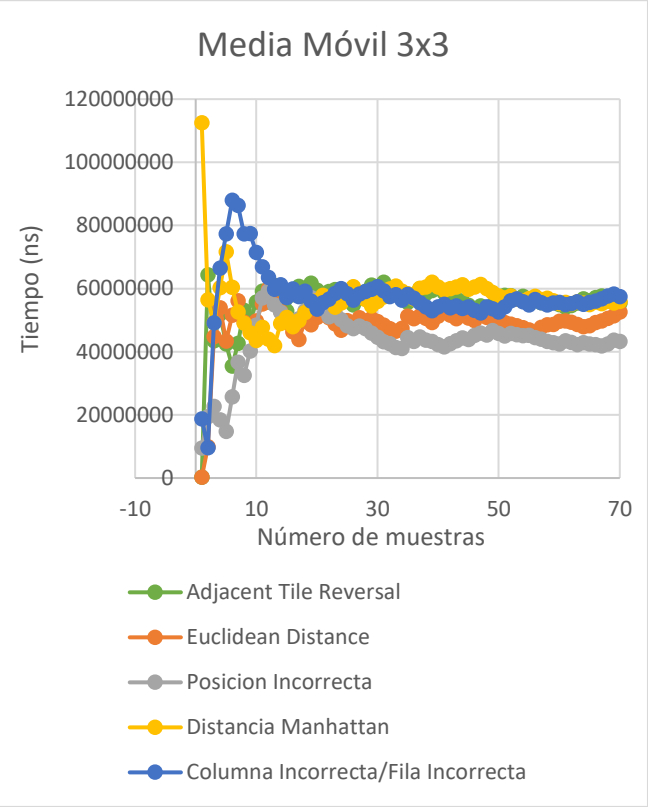

Gráfico 2

Se ha graficado la media móvil de todas las muestras hasta la actual, para ir suavizando las irregularidades.
Como se puede observar, en los tableros de 3x3 parece que la mejor heurística es la de "Posición Incorrecta".
El tiempo de media de resolución de esta heurística ha sido de 0,04s (4 centésimas de segundo). La media de las otras 4 heurísticas es de 0,055s, una diferencia del 21%.

Para la N=4, debido al gran tiempo que tardaba cada uno, solo se han realizado 20 ejecuciones por heurística. Los resultados han sido los siguientes:

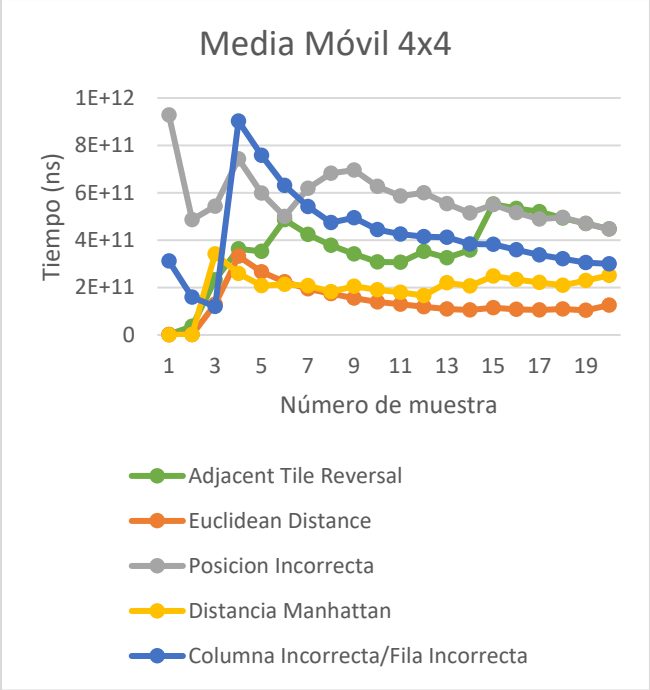

Gráfico 3

Como se puede observar, el tiempo de resolución ha aumentado en 100x al pasar a N=4. Parece ser que, esta N mayor, la mejor heurística pasa a ser la distancia euclídeana. Esta tiene un tiempo de resolución medio de 125s (2min 5s), mientras que las otras heurísticas tardan 361s de media (6min). Cabe destacar que la distancia de Manhattan también da buenos resultados, aunque no tan buenos como la Euclídea (4m 10s).

Como se puede observar, la tendencia de crecimiento respecto a la N (de centésimas de segundo a minutos solo con aumentar 1 el tamaño del problema) nos impide realizar tests de rendimiento para N iguales a 5 o superiores.

VIII. Guía de usuario

A continuación, se explicará como ejecutar el programa mostrando el flujo del programa de cara al usuario y todas las funcionalidades de las que dispone.

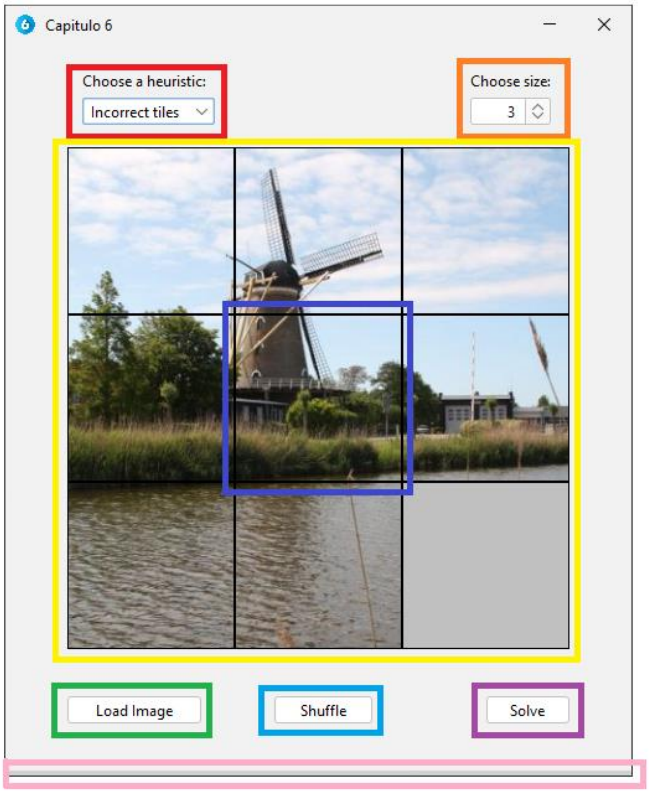

Ilustración 2: Estado inicial del programa

- **Rojo:** Selector que permite al usuario elegir que heurística usar entre las cinco implementadas
- **Naranja:** Selector que permite al usuario elegir el tamaño del puzle. Está delimitado entre 3 y 5.
- **Amarillo:** El puzle en sí, está formado por una imagen troceada en N*N trozos
- **Azul marino:** Casilla individual del puzle, se puede intercambiar por la casilla vacía si son contiguas
- **Verde:** Botón que abre un file chooser y permite que el usuario escoja otra imagen de fondo
- **Azul:** Botón que permite al usuario mezclar el puzle automáticamente
- **Lila:** Botón que ejecuta el algoritmo de resolución del puzle
- **Rosa:** Barra de progreso. Durante la ejecución del algoritmo de resolución estará en estado indeterminado. Una vez se haya resuelto y se proceda a enseñar la solución mostrará el progreso de la solución.

Al iniciar el programa nos encontraremos con una interfaz igual a la *Ilustración 3*. Desde este estado inicial podemos hacer 6 cosas diferentes: cambiar la heurística, cambiar el tamaño del tablero, cambiar la imagen del fondo, mezclar el tablero

manualmente, mezclarlo automáticamente y resolverlo.

I. **Cambiar la heurística**:
Al clicar en el botón de la heurística se nos presenta un selector que nos permitirá cambiar la heurística del algoritmo.

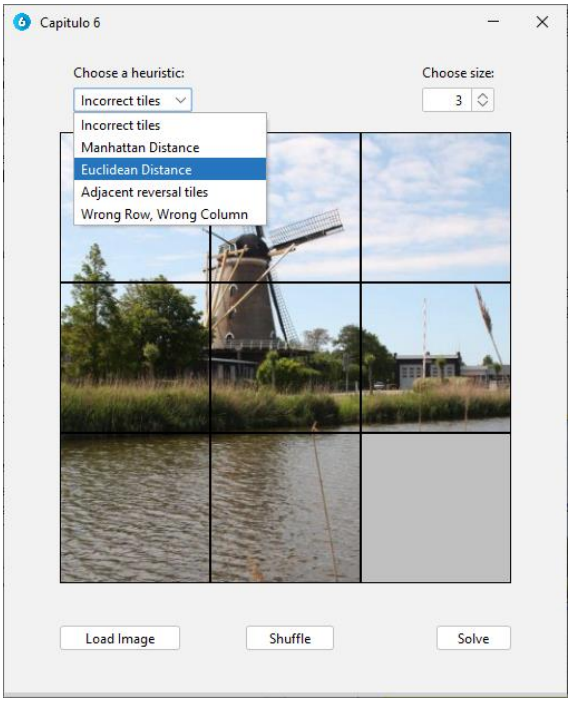

Ilustración 3

II. **Cambiar el tamaño:**
Al incrementar el tamaño del tablero podemos ver como la imagen se ha vuelto a fragmentar en N*N trozos

Ilustración 4

III. **Cambiar la imagen de fondo**
Al hacer clic en el botón de “Load image” se nos abrirá un File Chooser donde podemos cambiar la imagen del fondo.

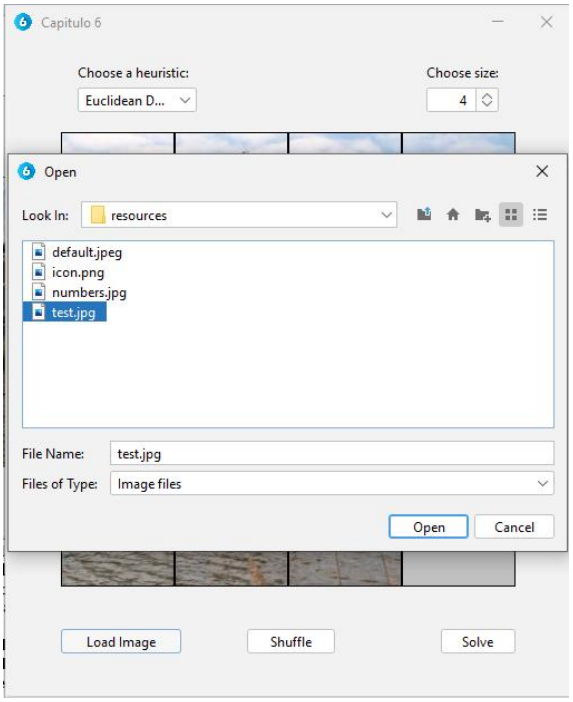

Ilustración 5

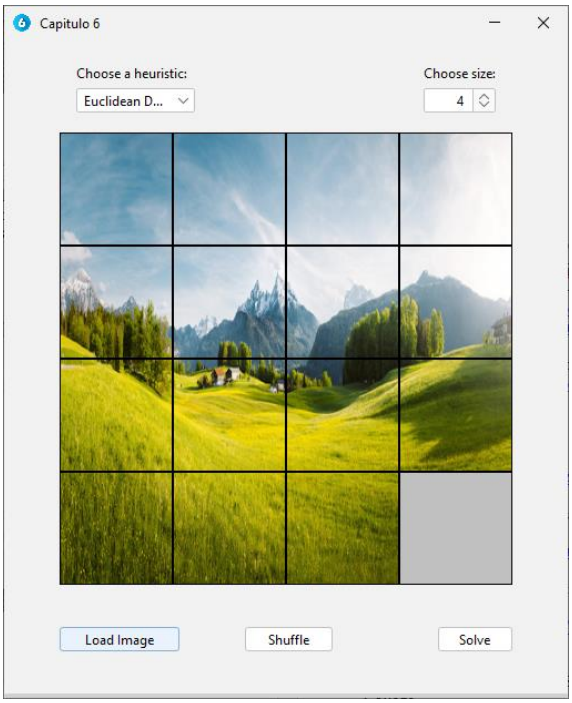

Ilustración 6

IV. **Mezclar el tablero manualmente**

Si hacemos clic en una casilla adyacente a la casilla vacía podemos intercambiar las posiciones.

Esto cumple dos funcionalidades: que el usuario pueda crear un estado personalizado y luego que el programa lo resuelva. O resolver el tablero manualmente.

En este ejemplo hemos llevado la casilla vacía de la esquina inicial a la contraria:

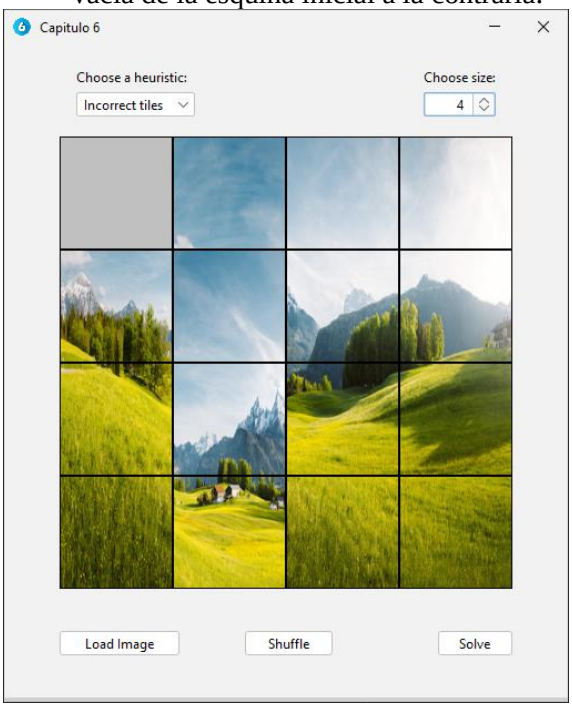

Ilustración 7

V. **Mezclar el tablero automáticamente**

El usuario también puede mezclar el tablero automáticamente con el botón "Shuffle"

Ilustración 8

VI. **Resolver el puzle**

Si el usuario hace clic en el botón de "Solve" el algoritmo se ejecutará y empezará a resolverlo.

Mientras lo resuelve la barra de progreso se pone en modo indeterminado como podemos ver en la *Ilustración 9*. Pero una vez ha encontrado la solución lo resuelve y en la barra de progreso podemos ver por qué porcentaje de la solución va (*Ilustración 10*).

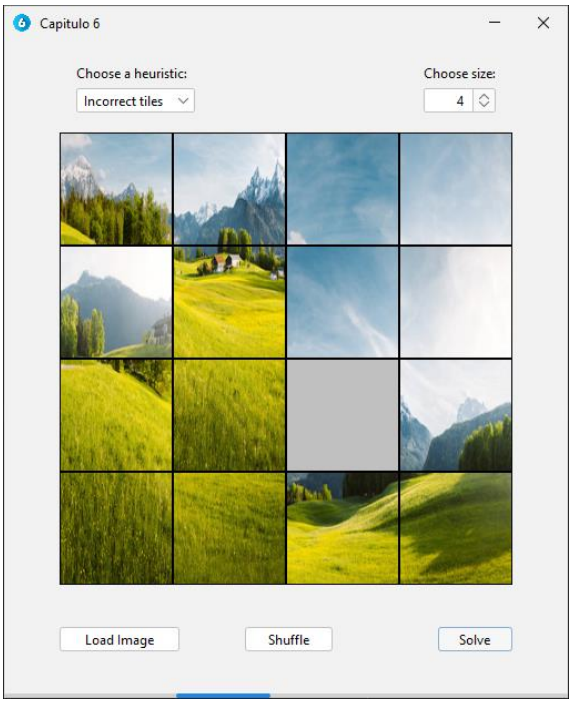

Ilustración 9

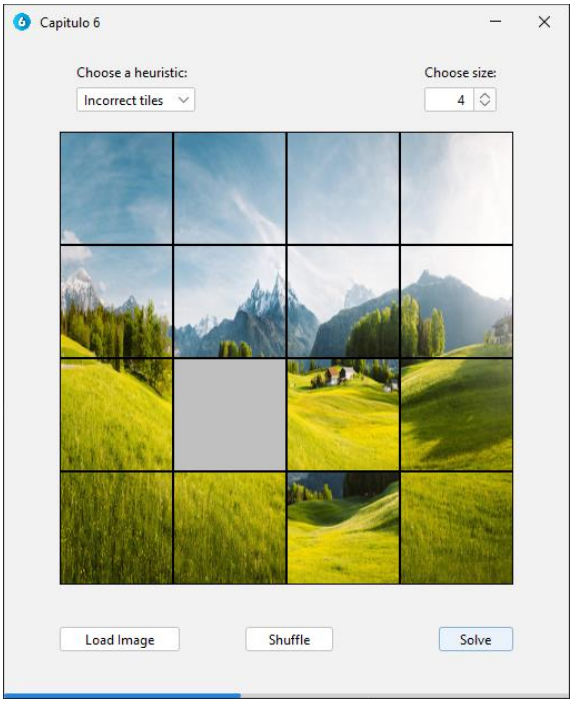

Ilustración 10

IX. CONCLUSIONES

Esta práctica nos ha permitido practicar los conocimientos adquiridos en clase y profundizar en los que adquirimos con anterioridad en otras asignaturas sobre los algoritmos de Branch and Bounding. Además de repasar el uso de heurísticas admisibles y su combinación.

Además, nos ha servido para continuar usando el patrón Modelo-Vista-Controlador y poder ver su utilidad como en las anteriores entregas.

Con esta práctica aprendido que para algunos problemas no se pueden generar estados iniciales aleatorios, ya que existen estados irresolubles. Para esto hemos implementado un algoritmo de mezcla basado en la generación de movimientos legales no repetitivos.

También nos ha dado juego para implementar distintas heurísticas para ver cuáles son mejores. Esto también nos ha dado pie a hacer distintos estudios de rendimiento para determinar que heurísticas son mejores.

Uno de los aspectos que nos han parecido interesantes a la hora de hacer la práctica ha sido a la hora de realizar el estudio sobre las heurísticas poder ver que dependiendo de la heurística el tiempo de ejecución puede variar de dos minutos a siete minutos y medio. Podríamos llevar a cabo otros estudios de tiempo de ejecución, como por ejemplo respecto el número de movimientos aleatorios a la hora de mezclar el tablero inicial.

En cuanto a tareas adicionales hemos llevado a cabo la funcionalidad de que el usuario pueda cambiar el tamaño del tablero, que pueda elegir una foto para ponerla de fondo y la animación de la solución. También hemos implementado un sistema de mezclado manual para que el usuario pueda generar un estado personalizado o resolver el estado aleatorio, dándole una doble funcionalidad al programa (resolutiva y lúdica). Finalmente cabe destacar que hemos implementado cuatro heurísticas más.

La práctica nos ha parecido una de las más interesantes de la asignatura.

X. BIBLIOGRAFÍA

15 Puzzle -- from wolfram MathWorld. (s. f.). Wolfram Mathword. https://mathworld.wolfram.com/15Puzzle.html

GeeksforGeeks. (2020, 10 junio). Difference between Backtracking and Branch-N-Bound technique. https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/

GeeksforGeeks. (2022). Branch and Bound Algorithm. https://www.geeksforgeeks.org/branch-and-bound-algorithm/

Gupta, S., & Bairagi, S. (s. f.). Evaluating Search Algorithms for Solving n-Puzzle. Sumitg. http://sumitg.com/assets/n-puzzle.pdf

Javatpoint. (s. f.). Branch and Bound. www.javatpoint.com. https://www.javatpoint.com/branch-and-bound

Latombe, J. (2011). Heuristic (Informed) Search. Standford University http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf

Marshall, J. (2005). Heuristic Search. Sarah Lawrence College http://science.slc.edu/%7Ejmarshall/courses/2005/fall/cs151/lectures/heuristic-search/

R. Kunkle, D. (2001). Solving the 8 Puzzle in a Minimum Number of Moves: An Application of the A* Algorithm. Massachusetts Institute of Technology. https://web.mit.edu/6.034/wwwbob/EightPuzzle.pdf

Solving the 8-Puzzle using A* Heuristic Search. (2009). IIT Kanpur https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf

XI. AUTORES

Jonathan Salisbury Vega, nació en Costa Rica en el 2000. A los 17 años se graduó de bachillerato en el instituto de educación secundaria IES Pau Casesnoves. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.

Joan Sansó Pericás, nació en Manacor en 2001. A los 18 años se graduó de bachillerato en el instituto de educación secundaria IES Manacor. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.

Joan Vilella Candia, nació en Elche en el 2000. A los 18 años se graduó de bachillerato en el instituto de educación secundaria IES Inca. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.

Julián Wallis Medina, nacido en Baleares en 2002. A los 17 años se graduó de bachillerato en el instituto de educación secundaria IES Son Pacs. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares a la vez que realiza prácticas en empresa de desarrollador Fullstack.