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  Julian Wallis Medina  *Universitat de les Illes Balears* Palma, España julian.wallis1@estudiant.uib.cat | Joan Vilella Candia  *Universitat de les Illes Balears* Palma, España joan.vilella1@estudiant.uib.cat |

*El coste asintótico de los algoritmos ha sido uno de los*

Keywords—MVC, rendimiento, coste asintótico, algoritmo.

# 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 puzzle 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.

# 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.

# Descripción del problema

# Solución Propuesta

# patrón mvc

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[[1]](#footnote-2) 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.

## Implementacion del Modelo

## Implementacion de la Vista

## Implementacion 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]](#footnote-3) 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.

### Programacion 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.

# suposiciones y posibles mejoras

# Estudios de Rendimiento

# Guia de usuario

# Conclusiones

# Bibliografía

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

[Branch and bound - javatpoint](https://www.javatpoint.com/branch-and-bound)

[Difference between Backtracking and Branch-N-Bound technique - GeeksforGeeks](https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/#:~:text=Branch%2Dand%2DBound%20involves%20a%20bounding%20function.&text=Backtracking%20is%20used%20for%20solving,used%20for%20solving%20Optimisation%20Problem.&text=In%20backtracking%2C%20the%20state%20space,until%20the%20solution%20is%20obtained.)

# 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 Inca. 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.

1. Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)