Capítulo 2

|  |  |  |
| --- | --- | --- |
| 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 |

*El coste asintótico de los algoritmos ha sido uno de los principales focos de estudio (y de problemas) a lo largo de la historia de la informática. Las soluciones más sencillas suelen tener costes asintóticos no deseables dependiendo del problema a resolver. Además, encontrar soluciones óptimas suele traer verdaderos quebraderos de cabeza a los distintos programadores. En este proyecto se ha implementado una aplicación capaz de visualizar los distintos tiempos que tienen estos algoritmos. Para ello se ha utilizado el Modelo Vista Controlador. Utilizar esta arquitectura permite un mayor control de errores del programa y facilidad de reutilización como de escalado. Todo esto a coste de una mayor complejidad de código. Con esta práctica se busca poner en práctica la arquitectura MVC y ofrecer una herramienta al programador para poder discernir de una manera gráfica con la que estudiar la viabilidad de los distintos algoritmos.*

Keywords—MVC, rendimiento, coste asintótico, algoritmo.

# Introducción

En este documento se pretende explicar como se ha obtenido una solución al problema propuesto y los distintos conceptos teóricos aplicados durante el proceso. También se pretende especificar la utilidad de la solución propuesta, así como de su implementación en un proyecto para poder visualizar los resultados.

# descripción del problema:

El problema en cuestión es una variante del recorrido de una pieza de ajedrez en un tablero de tamaño *NxN*. El problema original consiste en encontrar un camino en el que la pieza (usualmente un caballo) visite exactamente una vez cada casilla del tablero.

Este es un problema matemático utilizado comúnmente para aprender y aplicar conceptos de programación y desarrollo de algoritmos. En este caso se especifica que debe poder resolverse con seis piezas cada una con movimientos distintos y desde una posición inicial en el tablero arbitraria. El algoritmo también debe notificar en caso de que no exista una posible solución al problema dado.

El problema que se plantea en esta práctica es una extrapolación del conocido juego “El salto del caballo”. En el juego original, se ha de encontrar un recorrido (sin repetir casillas) desde una casilla inicial a otra casilla final, sin pasar por una ya visitada.

Como es fácil de observar, la descripción del problema encaja a la perfección con la definición del recorrido euleriano. Las casillas del tablero corresponderán con los nodos del grafo y las aristas serán los distintos movimientos que harán las piezas.

## Piezas disponibles

## Recorrido hamiltoniano

# Recursividad en la solucion

## Backtracking

## Complejidad de la solución

# Estructura mvc

## Implementación Modelo

## Implementación Vista

## Implementación Controlador

# Guía de usuario

# estudio de rendimiento

# conclusiones

# Recursividad

La recursividad es la forma en la cual un proceso se especifica basado en su misma definición. Más concretamente, en programación, es un método usual de resolver un problema de forma sencilla, como puede ser el problema de las [Torres de Hanoi](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), recorrido de Grafos, la búsqueda Euleriana…

Otros ejemplos de algoritmos que se definen de forma recursiva por su naturaleza son: los [Números de Fibonacci](https://en.wikipedia.org/wiki/Fibonacci_number), la [función factorial](https://en.wikipedia.org/wiki/Factorial).

A continuación, vamos a explicar los distintos tipos de recursividad.

## Tipos de recursividad

Dependiendo de la estructura de una función recursiva puede ser de los siguientes tipos:

### Recursividad Directa

La recursividad directa es cuando la función se llama a sí misma, pero tanto antes de la llamada recursiva como al volver, hay código que se ejecuta.

### Recursividad Indirecta

Se llama recursividad Indirecta cuando una función no se llama a sí misma, sino que esta desencadena una serie de llamadas de funciones, donde en una de estas llamadas se llama a la función original.

### Recursividad de cola

La recursividad de cola es un tipo de recursividad directa donde la llamada recursiva es la última sentencia de la función (todas las operaciones se hacen antes de la llamada recursiva).

### Recursividad de cabeza

Como la recursividad de cola, es un tipo de recursividad directa donde la llamada es la primera sentencia de la recursión. Es decir, las operaciones se realizan cuando la recursión ha llegado al caso base y las llamadas recursivas empiezan a volver.

### Recursividad Lineal

Es un tipo de recursión donde la cantidad de llamadas es proporcional al tamaño del problema (solo hay una llamada recursiva en la definición de la función, p. ej.: Factorial).

### Recursividad en Árbol o no Lineal

A diferencia de la Lineal, en estas funciones se hace más de una llamada recursiva, y, por ende, la recursión crece en forma de árbol (p ej.: Fibonacci, donde se hacen 2 llamadas recursivas en la definición de la función).

## Ventajas e inconvenientes de la Recursividad vs los algoritmos iterativos

### Ventajas

La recursividad puede **reducir la complejidad temporal**. Esto puede parecer contraintuitivo, pero si aplicas programación dinámica y **memorizas** soluciones parciales, puedes disminuir la complejidad temporal. Por ejemplo, para el cálculo de los números de Fibonacci, donde el algoritmo que memoriza es simplemente de complejidad tanto temporal como espacialmente (si se usa la estructura de datos adecuada).

Los programas recursivos suelen ser más “simples” y suelen tener menos líneas de código.

Son muy buenos para recorrer árboles. Debido a esto, se usan para implementar recorridos de árboles como el Preorden y para la generación de éstos (*véase Backtracking*).

### Desventajas

Si la cantidad de llamadas seguidas es muy grande puede dar problemas de memoria, ya que las llamadas van llenando la pila del sistema, y puede haber un error de *Overflow*, que será aún más rápido si se usan muchas variables locales, ya que en cada llamada recursiva éstas se tienen que guardar, ocupando aún más espacio.

Si se implementa mal, o se usa la recursividad en problemas sencillos, puede ser muy lenta. Esto es debido a que la recursividad necesita más tiempo al hacer llamadas y volver de estas debido a todo el manejo de memoria que se tiene que hacer, mientras que las iterativas son simples bucles sobre los mismos datos.

# Backtracking

El *backtracking* es un algoritmo general para encontrar soluciones a problemas que tienen que satisfacer un conjunto de condiciones.

La traducción literal del nombre seria “Paso atrás”, y se le da ese nombre debido a la naturaleza de estos algoritmos: van creando candidatos a la solución de manera incremental, y abandonan un candidato (dan un paso atrás) cuando se determina que ese candidato no podría ser solución.

De esta manera, se va generando un *árbol de soluciones parciales*, y el algoritmo **poda** la rama del árbol cuando ve que la solución parcial no podrá llegar a una solución completa.

El algoritmo puede ser programado para que pare al encontrar una solución, o puede ser creado de forma que encuentre todas las soluciones, que pare cuando haya pasado un tiempo determinado, o haya gastado una cantidad de recursos (ciclos de CPU, memoria) determinados.

## Ventajas e inconvenientes del Backtracking

Al ser un subconjunto de los algoritmos recursivos, se aplican las ventajas y desventajas de éstos. Adicionalmente, podemos enumerar:

### Ventajas

El backtracking, al ser un algoritmo que recorre todo el árbol de posibilidades, siempre va a ser capaz de encontrar todas solución, si es que existe alguna. De la misma manera, es capaz de asegurarte que no existe ninguna solución al conjunto de restricciones o condiciones establecidos.

Para según que tipos de problemas, es mucho más sencillo programar un algoritmo recursivo que intentar hacer uno iterativo para el mismo objetivo. Esto es debido a la facilidad de aplicar el concepto de recursión en problemas que necesitan satisfacer una lista arbitraria de condiciones, ya que puedes hacer que cada llamada recursiva intente encontrar una solución parcial a una de las condiciones, y si llegas a un estado que no podrá desarrollarse en solución final, vuelves atrás e intentas de nuevo.

### Inconvenientes

El coste asintótico de estas funciones suele ser del orden factorial () o exponencial (). Esto hace que para según qué problemas de gran tamaño, sea impracticable en el tiempo.

# descripción del problema

# Bibliografía

|  |  |
| --- | --- |
| [1] | Oracle, «UISwing Concurrency - Swing Worker,» [En línea]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html. |
| [2] | Wikipedia, «Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Divide-and-conquer\_algorithm. |
| [3] | «Oracle Java Documentation - SwingWorker,» [En línea]. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html. |
| [4] | J. M. Alarcón, «Rendimiento de Algoritmos y Big O - Campus MVP,» [En línea]. Available: https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx. |
| [5] | P. M. H. Thomas V. Perneger, «Writing a research article: advice to beginners - Oxford Academic,» [En línea]. Available: https://academic.oup.com/intqhc/article/16/3/191/1814554. |
| [6] | Wikipedia, «Recursion - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Recursion. |
| [7] | Wikipedia, «Backtracking - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Backtracking. |