Objetivos

- Repasar el concepto de Vuelta Atrás visto en teoría.
- Estudiar distintos casos en los que se aplica esta técnica.
- Implementar algoritmos de Vuelta Atrás para resolver problemas concretos.

Conceptos

Dentro de las técnicas de diseño de algoritmos, el método de Vuelta Atrás (del inglés **Backtracking**) es uno de los de **más amplia utilización en problemas basados en optimización**.

Hasta ahora hemos aprendido técnicas que nos permiten encontrar una solución de manera eficiente, tales como aquellas basadas en algoritmos voraces. Estos algoritmos se basan en encontrar una buena solución, no la mejor, en el menor tiempo posible. Así, se sacrifica calidad de resultados por eficiencia y rapidez de ejecución. Pero **existen muchos casos** en los que la **única manera de encontrar una buena solución pasa por revisar de manera exhaustiva todas las posibles combinaciones de resultados** hasta encontrar uno válido o un conjunto de posibles resultados válidos.

El **problema tipo** que vamos a resolver tiene las siguientes características: sea un determinado **problema algorítmico que plantea una solución por etapas**, es decir, la solución puede ser expresada como una n-tupla de valores $[x_1, x_2, \ldots, x_n]$. A su vez, el problema proporciona un determinado número de posibles valores u opciones, m, que pueden tomar los elementos que forman la tupla. El objetivo será encontrar que combinación de esos m valores, para cada uno de los elementos que forman el resultado final, proporciona la mejor solución al problema. Muchas veces dicha solución estará formada no solo por una posible combinación de valores de la tupla sino por un conjunto de posibles combinaciones.

Revisar de manera exhaustiva todas las posibles combinaciones de valores para los N elementos de la tupla puede resultar muy ineficiente. Es por ello que **uno de los primeros pasos a la hora de trabajar con un algoritmo de vuelta atrás es encontrar un conjunto de restricciones aplicables a las posibles soluciones al problema**. Este conjunto de restricciones suele extraerse del enunciado del problema y nos ayudará a reducir el espacio de búsqueda y, así, a disminuir el tiempo de ejecución de nuestro algoritmo. Una vez fijadas las restricciones hemos de desarrollar un recorrido que irá pasando a través de las etapas necesarias para construir la solución.

Podemos **resumir cada uno de los pasos** a tener en cuenta en un algoritmo **de Vuelta Atrás** de la siguiente manera:

1. **Identificar** solución formada por una tupla de n elementos $[x_1, x_2, \ldots, x_n]$. Para cada uno de esos elementos existirá un **conjunto de posibles valores**.
2. Obtener del enunciado un conjunto de **restricciones impuestas al resultado final** que nos ayudará a disminuir el espacio de búsqueda y a conocer si una solución es o no válida.
3. El proceso consta de **N etapas** (tantas como elementos formen la solución final) y **cada una de esas etapas estará representada por una llamada recursiva**. En cada una de esas etapas, k se desarrollará un bucle en el que iremos probando con los posibles valores que puede tener el elemento de la solución que ocupa la posición k, x_k. En dicho bucle se llevan a cabo las siguientes acciones:
    a. Se **escoge un valor para** x_k y se comprueba si la nueva solución parcial formada es o no válida. Nos ayudaremos de las restricciones extraídas en el paso 2.
    b. Si la solución parcial **NO es válida**, se prueba con otro valor para x_k y se vuelve al paso anterior.
    c. Si la solución parcial **SI es válida y**, además, **llegamos a una solución final**, el problema habrá finalizado.
    d. Si la solución parcial **SI es válida pero NO forma la solución final**, pasamos a la siguiente etapa, $k+1$, mediante una llamada recursiva, utilizando la solución parcial encontrada como parámetro de entrada. Es decir, volvemos al paso 3. Si esta llamada recursiva tiene éxito, es decir, se encuentra la solución final, habremos acabado. Si no tiene éxito, se desmarca el valor escogido para x_k y se vuelve al paso a.

Una etapa finaliza cuando se ha llegado a la solución final o se han probado todos los posibles valores para x_k. Si después de probar todos los posibles valores de x_k no se encuentra una solución válida, la llamada recursiva para esa etapa habrá fracasado. En ese momento se lleva a cabo la vuelta atrás, que tiene como resultado volver a la etapa anterior, $k-1$, para probar con otro valor para el elemento x_{k-1} de la tupla.

Tal y como observamos, **la recursividad tiene un papel importante en la implementación de estos algoritmos**. El siguiente algoritmo general resume los pasos comentados anteriormente.

```
  int VueltaAtras(int etapa) {
1     int exito=0;
2     // Inicialización de variables y opciones
3     while (!(exito) && HayaOpciones()) {
4           SeleccionarNuevaOpcion();
5           AnotarOpcion();
6           if (SolucionFinalVálida()) {
7                 exito=1;
8                 MostrarResultado();
9           } else if (SoluciónParcialVálida()) {
10                exito=VueltaAtras(etapa + 1);
11                if (!exito) {
12                      CancelarAnotacion();
13                }
14          }
15    }
16    return exito;
  }
```

En este algoritmo genérico podemos observar cada uno de los pasos comentados anteriormente. El bucle del paso 3 comienza en la línea 3. Este bucle seguirá ejecutándose mientras no se haya encontrado la solución final y queden posibles valores que probar para el elemento de la etapa k. Cada nueva opción es anotada, estableciéndose los cambios necesarios en la solución parcial y en la línea 6 se comprueba si esa nueva solución parcial constituye un resultado final válido. Si es así, se informa la variable éxito con un valor igual a true y se muestra el resultado (líneas 7-8). Otro caso posible es que no sea una solución final pero sí una solución parcial válida (línea 9). En ese caso hacemos una llamada recursiva, con la solución parcial formada, con el objetivo de completar la siguiente etapa (línea 10). Si esta llamada tiene éxito, habremos terminado. En caso contrario, se cancela la anotación realizada al principio del bucle (línea12) y volveremos a probar con otra nueva opción (línea 4). Si el bucle finaliza porque hemos pasado por todas las opciones sin haber encontrado la solución final, esta llamada recursiva acabará y en la llamada anterior a la función, la que se encarga de la etapa $k-1$, se tendrá que probar con otra nueva opción ya que la llamada recursiva realizada para completar la etapa k no tuvo éxito (líneas 11 y 13). Eso es lo que llamamos la VUELTA ATRÁS.

De la **calidad de las restricciones impuestas** para comprobar la validez de las soluciones parciales dependerá **el resultado final y la eficiencia del algoritmo**.

Finalmente, indicar que el **proceso de búsqueda de soluciones** comentado anteriormente **se asemeja a la construcción de una estructura con forma de árbol y a su recorrido en profundidad**.

Experimentos

E1. (60 min.). Laberinto. Una matriz bidimensional cuadrada (N x N) puede representar un laberinto cuadrado. Cada posición contiene un 0 que indica que la casilla es transitable o un 1 si no lo es. La entrada y salida del laberinto serán representadas por casillas siempre transitables y que pueden variar de una ejecución a otra. Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida.

Como la mayoría de algoritmos de vuelta atrás, será un proceso por etapas en las que vayamos probando diversos caminos.
El código propuesto para solucionar este problema es el siguiente:

```
private static final int TAM = 5;
private static final int NMOV = 4;
private static final Posicion[] MOVIMIENTOS=
    {new Posicion(-1,0),new Posicion(1,0),new Posicion(0,-1),new Posicion(0,1)};



// Métodos auxiliares
boolean valida(Posicion p) {
        return (0 <= p.getFila()
                 && p.getFila()<TAM
                         && 0<=p.getColumna()
                         && p.getColumna()<TAM);
}

boolean usada(Posicion p, int[][] laberinto) {
        return(laberinto[p.getFila()][p.getColumna()]==9);
}


boolean obstaculo(Posicion p, int[][] laberinto) {
        return(laberinto[p.getFila()][p.getColumna()]==1);
}

void marcar(Posicion p, int[][] laberinto) {
        laberinto[p.getFila()][p.getColumna()]=9;
}

void desmarcar(Posicion p, int[][] laberinto) {
        laberinto[p.getFila()][p.getColumna()]=0;
}


boolean llegamos(Posicion ent,Posicion sal) {
        return(ent.getFila()==sal.getFila() && ent.getColumna()==sal.getColumna());
}



// Método recursivo que implementa el backtracking
boolean laberinto_bt(int[][] laberinto,Posicion entrada, Posicion salida) {
        boolean exito = false;
        int m=0;


        if (llegamos(entrada,salida))
                        exito=true;
        else {
                        Posicion pos_actual = new Posicion(0,0);
                        while (m<NMOV && !exito) {
                            marcar(entrada,laberinto);
                            pos_actual.setFila(entrada.getFila()+MOVIMIENTOS[m].getFila());
                                                        pos_actual.setColumna(entrada.getColumna()
+MOVIMIENTOS[m].getColumna());
                             if (valida(pos_actual)
                                 && !usada(pos_actual,laberinto)
                                 && !obstaculo(pos_actual,laberinto)) {
                                 exito = laberinto_bt(laberinto,pos_actual,salida);
                                 desmarcar(pos_actual,laberinto);
                            }
```

```
                m++;
            }
        }
        return exito;
    }
```

Utilizando dicho código se proponen los siguientes ejercicios:

a) Realice pruebas y compruebe la salida del programa en función de distintas entradas.

b) Modifique el programa para que se muestre por pantalla el camino correcto que vamos recorriendo por el laberinto.

c) Modificar el programa para que se muestre por pantalla, además de la información del apartado anterior:

1. Posición en la que nos encontramos.
2. Posición nueva que va a ser inspeccionada.
3. Si la posición no es válida, que se muestre el motivo por el cual no lo es (fuera rango de la matriz, obstáculo o casilla ya utilizada).
4. Si la posición es válida pero no se ha encontrado un camino válido partiendo de ella, mostrar un mensaje indicando que no es posible avanzar desde ahí.

d) Modifique el programa para que, después de que la función laberinto sea llamada desde el experimento, se muestre por pantalla como queda finalmente el laberinto. Es decir, se ha de mostrar en pantalla el laberinto con las casillas que forman el camino válido marcadas.

Por ejemplo:

0	1	0	1	1
0	0	0	0	1
0	1	1	0	1
0	0	0	0	1
0	1	1	0	0

9	1	0	1	1
9	0	0	0	1
9	1	1	0	1
9	9	9	9	1
0	1	1	9	9

Problemas

P1. (60 min). El problema de las 8 reinas. Disponemos de un tablero de ajedrez de tamaño 8x8 y se trata de colocar en él ocho reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren dos reinas ni en la misma fila, ni en la misma columna, ni en la misma diagonal.

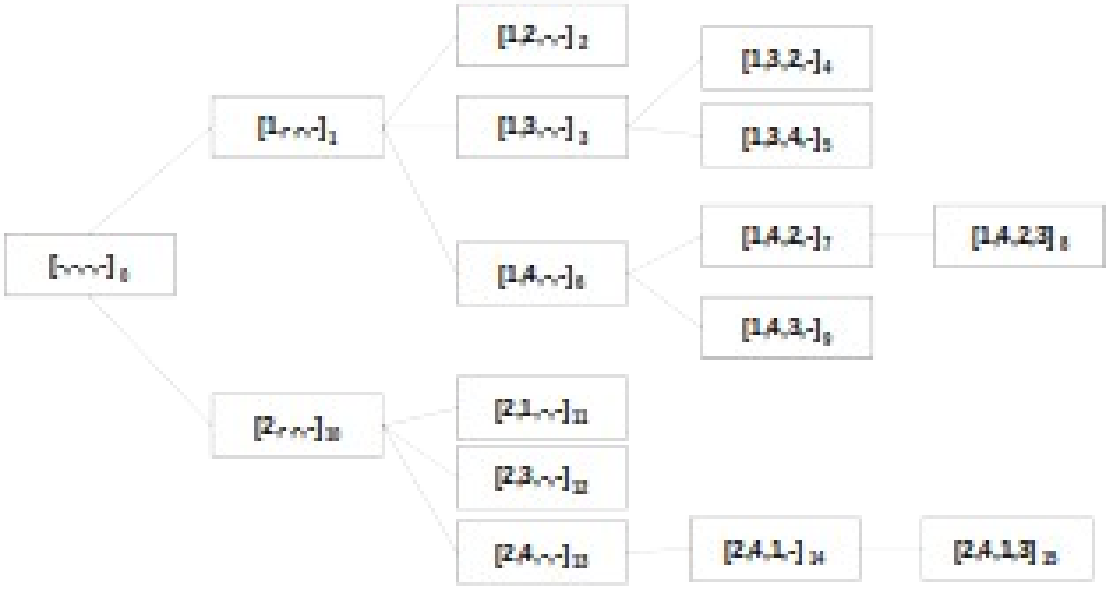
Numeramos las reinas del 0 al 7. Cualquier solución a este problema estará representada por una 8-tupla $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ en la que cada x_i representa la columna donde la reina de la fila i-ésima es colocada. Una posible solución al problema podría ser la tupla $[3,5,7,1,6,0,2,4]$ (reina en fila 0 colocada en columna 3, reina en fila 1 colocada en columna 5, etc.).

Lo primero que hay que hacer es definir las restricciones que nos van a ayudar a disminuir el espacio de búsqueda:

1. Los valores que pueden tomar cada una de las posiciones de la tupla. En nuestro caso los valores permitidos son $S = [0,1,2,3,4,5,6,7]$. De todos los números posibles hemos restringido la búsqueda a solo 8 valores.

2. El enunciado nos dice que ninguna reina puede estar ni en la misma fila, ni en la misma columna, ni en la misma diagonal. Si se observa, las dos primeras restricciones ya se cumplen si organizamos el resultado como un vector cuyo índice representa a las 8 filas y además indicando que en el vector no podrán existir números repetidos. Todavía nos queda definir la última

restricción: dos reinas no pueden encontrarse en la misma diagonal. Sean (x_1, y_1) y (x_2, y_2) las coordenadas de dos reinas, la tercera restricción se cumple si: $|x_1 - x_2| \neq |y_1 - y_2|$. De esta manera, y aplicando las restricciones, en cada etapa k iremos generando sólo las k-tuplas con posibilidad de solución. Hemos comentado en la teoría que el recorrido llevado a cabo por este tipo de algoritmos se asemeja a la creación y recorrido en profundidad de un árbol. Veamos un ejemplo con un tablero de 4×4 y con un conjunto válido de valores igual a $S = [1, 2, 3, 4]$:



En la anterior figura se muestra un árbol cuya raíz representa el elemento $[\ ,\ ,\ ,\ ]$. Al lado de cada nodo del árbol hay un número que nos indica el orden en que se generó. Como podemos observar, por cada etapa k de comprueba si un determinado valor x_k sirve de base para una buena solución. A partir de dicho nodo se van generando más nodos y ramas del árbol. Si encontramos una solución, perfecto. Si no, en el momento que, gracias a las restricciones, veamos que hemos llegado a un resultado no válido, volveremos hacia atrás y empezaremos de nuevo en el mismo nivel del árbol.

Por ejemplo, Una vez generado en tercer lugar el nodo $[1,3,\ ,\ ]$ comprobamos si es una solución válida pasando a la siguiente etapa y eligiendo dos posibles valores: $[1,3,2,\ ]$ y $[1,3,4,\ ]$. Llegados a este punto comprobamos que no pueden ser soluciones buenas ya que incumplen la restricción de la diagonal. Así que hemos de volver hacia atrás y probar con $[1,4,\ ,\ ]$.

a. Implemente el problema de las 8 reinas usando un algoritmo recursivo. El programa acabará cuando se encuentra una solución válida.

   Idea: sería útil contar con dos funciones auxiliares. Una para saber si un resultado parcial es válido, es decir, que compruebe si se dan o no las restricciones. Para ello habría que comprobar si el nuevo valor añadido al vector resultado en la etapa k implica que dicha reina está o no en la misma diagonal que alguna de las anteriores. Y otra función para calcular el valor absoluto de un número (para la restricción de la diagonal). Utilice el esquema suministrado en la parte teórica como base para el algoritmo.

b. Modifique el algoritmo anterior para que encuentre todas las posibles soluciones válidas.

P2 (90 min) Los subconjuntos de una suma dada. Sea W un conjunto de enteros no negativos y M un número entero positivo. El problema consiste en diseñar un algoritmo para encontrar un posible subconjunto de W cuya suma sea exactamente M.

En primer lugar, podemos suponer que el conjunto viene dado por un vector de enteros no negativos y que ya se encuentra ordenado de forma creciente. Con esto en mente, la solución al problema podremos expresarla como una n-tupla $X=[x_1, x_2, \ldots x_n]$ en donde x_i podrá tomar los valores 1 ó 0 indicando que el entero i forma parte de la solución o no. El algoritmo trabaja por etapas y en cada etapa ha de decidir si el k-ésimo entero del conjunto interviene o no en la solución.
A modo de ejemplo, supongamos el conjunto $W=[2,3,5,10,20]$ y sea $M=15.$ Existen dos posibles soluciones, dadas por las 5-tuplas $[1,1,0,1,0]$ y $[0,0,1,1,0].$ Una vez adaptada la solución final a la técnica de vuelta atrás, el siguiente paso es determinar las restricciones que nos harán disminuir el espacio de búsqueda:

1. Estamos en la etapa $k.$ Si a la suma de valores que hemos conseguido hasta esa etapa, añadimos el siguiente entero de la lista W y superamos el valor $M,$ querrá decir que el camino escogido no es una buena opción.

   Por ejemplo:

   $[1,1,1, \ , \ ],$ en la etapa $k=3,$ la suma parcial conseguida hasta ahora es de 10. Si a esta suma le añadimos el siguiente valor posible, el 10, conseguiremos un valor que sobrepasa a M, por lo cual no es un buen camino y tendremos que hacer vuelta atrás. Esta restricción es posible debido al orden creciente de la lista de enteros.

2. Estamos en la etapa $k.$ Si a la suma de valores que hemos conseguido hasta esa etapa se le suma el resto de números enteros de la lista y el resultado es menor que M, tampoco es un buen camino.

   Por ejemplo:

   Supongamos que $W=[2,3,5,6,7]$ y que en la etapa $k=3$ tenemos la siguiente solución parcial $[0,0,0, \ , \ ].$ Si a la suma que conseguimos con esta solución (0) le sumamos los otros dos números restantes $(0+6+7=13)$ el resultado será menor que el valor $M=15,$ por lo cual por este camino no conseguiremos nada.

A continuación, os mostramos un ejemplo del árbol que se construye mediante la generación de las etapas y la vuelta atrás. El ejemplo está basado en un vector $W=[2,3,5,10,20]$ y un valor $M=15.$

a. Implementemos este algoritmo de manera recursiva. Para ello utilizaremos los siguientes elementos:

- v: vector de enteros positivos en orden creciente de tamaño N
- m: valor a encontrar
- k: etapa en la que nos encontramos
- x: vector con el resultado final formado por ceros y unos
- s: suma de los elementos según el resultado parcial que llevamos hasta ahora
- r: suma del resto de elementos del vector W sin explorar

b. Modifique el anterior programa para encontrar todos los posibles subconjuntos de W cuya suma sea exactamente M.

Bibliografía Básica

1. **Documentación Oficial Oracle JAVA**. https://docs.oracle.com/en/java/index.html
2. **Fundamentos de Algoritmia**. G. Brassard, P. Bratley. Prentice Hall.
3. **Introduction to Algorithms**. TH. Cormen, CE. Leiserson, RL. Rivest, C. Stein. MIT Press, 2001.
4. **Técnicas de Diseño de Algoritmos**. R. Guerequeta, A. Vlleillo. Universidad de Málaga/Manuales, 1997. Capítulo 6, páginas 211-254.