Práctica 2

Asignatura:

Programación y Estructuras de Datos Avanzadas

Iker Domingo Pérez
71126454N
UNED Madrid - Las Tablas
Curso 2016/2017

Las ocho reinas

1.- Describa el esquema algorítmico utilizado y cómo se aplica al problema.

El esquema algorítmico es el recomendado en el enunciado de la práctica: Vuelta Atrás.

Nos enfrentamos al problema de recorrer en profundidad un grafo de tamaño variable donde no todos los caminos son válidos. La vuelta atrás se utiliza junto con la poda para reducir el coste de esta exploración.

A lo largo de esta práctica pondremos el ejemplo de un tablero de dimensiones 8x8, pero el ejecutable resuelve este problema para tableros cuadrados o rectangulares de cualquier dimensión. Mencionar también que en nuestro programa los lados del tablero serán las variables globales ***int n*** e ***int m***. Cuando tengamos un rectángulo, ***n*** será el menor de los lados.

El problema pide que situemos tantas reinas como podamos en un tablero sin que se ataquen entre ellas. Para ello, las reinas no pueden coincidir en la misma fila, ni en la misma columna, ni en sus diagonales. En los tableros cuadrados se podrán colocar tantas reinas como filas tenga el tablero, en los rectangulares se colocarán tantas como la longitud del lado más corto.

Así pues, un grafo está compuesto por ramas y nodos, o aristas y vértices, o en nuestro caso por filas y posiciones. En nuestro tablero-grafo de 8x8 tenemos 8 filas, y en cada fila 8 posiciones. Vamos a ir explorando fila a fila, desde la primera posición hasta que encontremos una posición que añadir a la secuencia *k-prometedora*. Cuando tengamos esta posición *Completable()* la guardaremos en un vector y pasaremos a la siguiente fila para *IniciarExploraciónNivel()*.

La vuelta atrás se implementa en dos momentos del algoritmo:

- Cuando tenemos una *SoluciónCompleta()*.
- Tras explorar una fila sin encontrar una posición válida, !Completable().

En el primero de los supuestos, cuando tenemos una *SoluciónCompleta()*, se introduce la vuelta atrás para seguir explorando el grafo desde la última fila visitada que aún tenga posiciones

sin visitar, *OpcionesPendientes()*. De esta forma, estamos en la fila 8 y tenemos una *SoluciónCompleta()*, borramos la posición guardada de la fila 8 y volvemos a la fila 7. Si la fila 7 tiene guardada la última posición (no le quedan posiciones por visitar) entonces volvemos a la fila 6 que, por la naturaleza de este problema (no pueden coincidir 2 reinas en la misma columna), estamos 100% seguros que no estará en la última posición y por tanto aún tendrá *OpcionesPendientes()*.

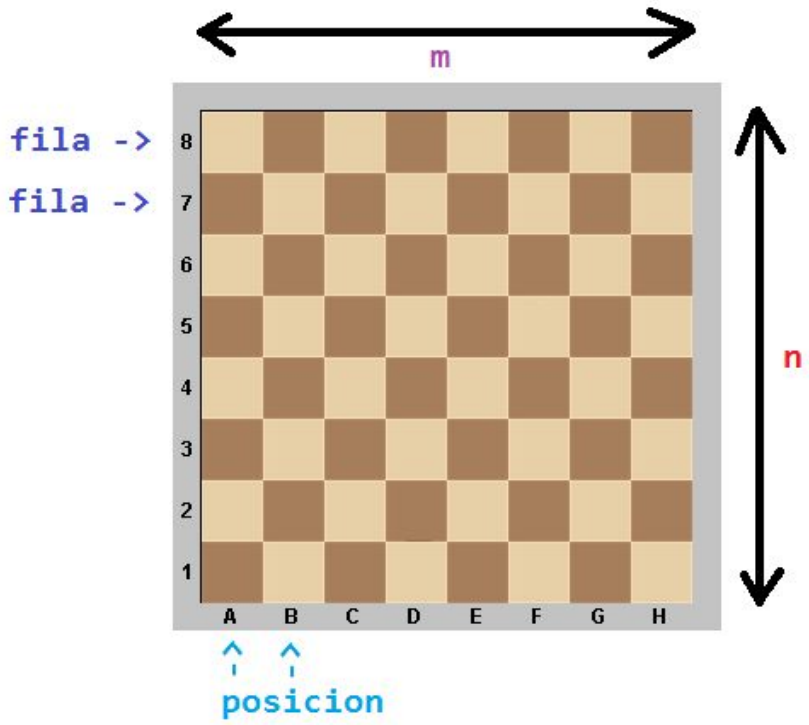
En el segundo de los casos, al igual que antes, supongamos que hemos recorrido las 5 primeras filas y tenemos una secuencia *5-prometedora* (A1, B3, C5, D2, E4) estamos analizando la fila 6 y llegamos a la última posición sin un acierto. En este caso volvemos atrás a la fila 5 y analizamos las posiciones que quedan sin explorar: E5, E6, E7…

En este ejemplo hemos utilizado no sólo la vuelta atrás sino también la poda. Cuando estábamos analizando la fila 6, teníamos una secuencia *5-prometedora* formada por las 5 posiciones guardadas, pero en la fila 6 no hemos encontrado una posición que hiciese esta solución *Completable()*, por lo que hemos podado la rama A1-B3-C5-D2-E4 y hemos vuelto atrás.

La poda es el método que ofrece este esquema algorítmico para reducir el coste computacional. Hemos comprobado que la solución parcial que teníamos no era *Completable()*, por lo que no tenía sentido seguir avanzando por ese camino.

Antes de entrar a analizar los métodos y el código del programa es necesario explicar la estructura y las variables utilizadas.

```
for (int i=fila; i<n; i++) {
   for (int j=posicion; j<=m; j++) {
```



La estructura el programa no es la de un algoritmo recursivo típico donde se tiene un método que analiza una fila, y al terminar vuelve a llamarse a sí mismo para analizar la siguiente fila. El algoritmo está construido en torno a dos bucles *for* anidados dentro de un bucle infinito *while(true)* que recorren las filas (n) y las posiciones (m) del tablero de ajedrez. Para poder implementar la vuelta atrás hemos utilizado dos variables globales (***int*** *fila* e ***int*** *posicion*) como los índices de estos bucles. De forma que cuando sea necesario hacer la vuelta atrás será suficiente con modificar los índices y volver a entrar a los bucles.

El algoritmo arranca explorando cada nivel, en cada posición comprueba si ofrece una solución *Completable()*. Si, por ejemplo, la posición 2 de la fila 5 ofrece una solución *Completable()* este 2 se almacena en la posición 5 (realmente se almacena en la posición 4) de un vector de dimensión *n*. Como hemos dicho antes, para tableros rectangulares se podrán colocar tantas reinas como la longitud del lado más corto, que en este caso será igual a *n*, o lo que es lo mismo el número de filas. Por tanto, el vector/array con las soluciones deberá tener longitud *n*.

Ya tenemos el 2 almacenado en la posición 5 del vector de soluciones, que en nuestro programa es ***int****[n] tablero*. En este punto no hace falta seguir explorando el nivel, sino que hay que pasar al siguiente nivel, para ello cuando se encuentra un acierto (*ComprobarSolucion(fila, posicion)* devuelve ***true***) se modifica el índice *posicion* dándole el valor 1 y se sale del bucle interno con ***break;*** esto provoca que el bucle externo se autoincremente (siguiente fila) y vuelva a entrar al bucle interno para *IniciarExploraciónNivel()* desde la posición 1.

```
for (int i=fila; i<n; i++) {
    for (int j=posicion; j<=m; j++) {
        if (ComprobarSolucion(i, j)) {
            tablero[i]=j;
            fila++;
            posicion=1;
            break;
        }
```

Con esta estructura se analiza cada posición hasta encontrar una *Completable()*. Para comprobar que hemos explorado todas las filas y, por tanto, tenemos una *SoluciónCompleta()*, lo que hacemos es simplemente preguntar en qué fila estamos. Si estamos en la última fila (*n-1*) imprimimos la solución.

```
if (i==n-1) {
    ImprimirSolucion();
    index++;
```

** ***index*** *es una variable global que lleva la cuenta de las soluciones encontradas* **

El siguiente paso es realizar la vuelta atrás. Para ello preguntamos qué posición tiene guardada la última fila *(tablero[i-1] = ?*), para saber si le quedan *OpcionesPendientes()*. Si le quedan *OpcionesPendientes()* se invoca *Vuelta1(i)*, si no, *Vuelta2(i)*.

```
if (tablero[i-1]==m) {
    Vuelta2(i);
    break;
}   Vuelta1(i);
    break;
```

El método *Vuelta1(i)* modifica el índice *fila* restándole uno (la próxima entrada al bucle será en la fila anterior). Modifica también el índice *posicion* y lo sitúa en el siguiente punto que tiene que explorar (*posicion* = *tablero[i-1]* + *1*). Por último elimina las posiciones que estuviesen guardadas en el array de soluciones para la fila actual y la anterior.

```
public static void Vuelta1(int i) {
    fila = i - 1;
    posicion = tablero[i-1] + 1;
    tablero[i] = tablero[i-1] = 0;
    scan++;
}
```

** ***scan*** *lleva la cuenta de las veces que se realiza la vuelta atrás* **

El método Vuelta2(i) realiza la misma operación, en esta ocasión para la fila antepenúltima.

```
public static void Vuelta2(int i) {
    fila = i - 2;
    posicion = tablero[i-2] + 1;
    tablero[i] = tablero[i-1] = tablero[i-2] = 0;
    scan++;
}
```

El siguiente punto es saber cuándo hemos terminado de explorar todas las posibles soluciones del tablero. Teniendo en cuenta que es un bucle que va de la primera a la última fila, de la primera a la última posición, se habrá terminado de explorar cuando en la primera fila tengamos guardada la última posición (*tablero[0]* = *m*) y estemos explorando la última posición (*j* = *m*) de la segunda fila (*fila* = *1*). Esta comprobación se realiza al inicio de la exploración.

```
if (tablero[0]==m&&fila==1&&j==m) {
    System.exit(0);
}
```

Hasta aquí el tratamiento de los aciertos y las soluciones. Cuando se produce un fallo no se realiza ninguna acción especial, sino que se deja que el bucle continúe y explore la siguiente posición. Lo que sí se comprueba es si el fallo ha ocurrido en la última posición de una fila. Esto indica que no quedan *OpcionesPendientes()*, indica que la solución *Completable()* que teníamos hasta ahora no sirve y tenemos que podar y volver atrás. Por tanto, si estamos en la última posición de una fila (*j* = *m*) invocamos la vuelta atrás de la misma forma que hicimos cuando teníamos una solución completa: preguntando qué posición tiene guardada la fila anterior

```
if (j==m) {
    if (tablero[i-1]==m) {
        Vuelta2(i);
        break;
    }   Vuelta1(i);
        break;
}
```

Por último, cuando salimos del bucle interno, se comprueba si se ha modificado el índice *fila* del bucle externo. Este índice se modifica en los métodos *Vuelta1(i)* y *Vuelta2(i)*. Si ha sido modificado se lanza un ***break;*** para salir del bucle externo y volver a entrar para escanear la siguiente fila.

```
if (fila<i) break;
```

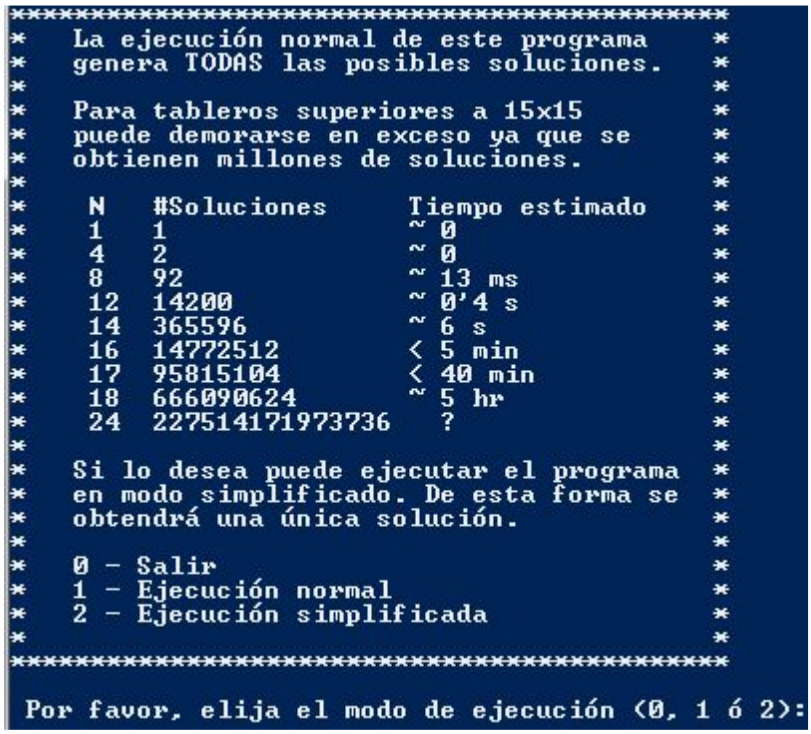
El método principal *VueltaAtras()*, núcleo del algoritmo, queda así:

```
public static void VueltaAtras () throws NullPointerException, IOException {
    do {
        for (int i=fila; i<n; i++) {
            for (int j=posicion; j<=m; j++) {
                if (tablero[0]==m&&fila==1&&j==m) System.exit(0);
                if (ComprobarSolucion(i, j)) {
                    tablero[i]=j;
                    if (i==n-1) {
                        ImprimirSolucion();
                        index++;
                        if (modoSimple == true) System.exit(0);
                        if (tablero[i-1]==m) {
                            Vuelta2(i);
                            break;
                        }   Vuelta1(i);
                            break;
                    }
                    fila++;
                    posicion=1;
                    break;
                }
                if (j==m) {
                    if (tablero[i-1]==m) {
                        Vuelta2(i);
                        break;
                    }   Vuelta1(i);
                        break;
                }
            }
            if (fila<i) break;
        }
    } while (true);
}
```

En este fragmento de código aparece un método del que no hemos hablado, *ComprobarSolucion(fila, posicion)* y el boolean *modoSimple*.

Este boolean se utiliza cuando se ejecuta el programa introduciendo un valor superior a 15. Se pregunta al usuario si quiere ejecutar el programa de la forma normal, exponiéndose a una ejecución prolongada para sacar todas las posibles soluciones, o si prefiere el modo simplificado, donde se finaliza el programa al encontrar una solución.

(En el último apartado de este documento se analiza con más detalle el modoSimple.)



El método *ComprobarSolucion(fila, posicion)* es el encargado de validar una posición. Para ello compara la posición que se está evaluando con las posiciones guardadas en el array de soluciones.

```
public static boolean ComprobarSolucion (int fila, int posicion) {
    if (fila == 0 && tablero[0]==0) return true;
    for (int i=0; i<fila; i++) {
        if (posicion == tablero[i] ||
            posicion == tablero[i]-(fila-i) ||
            posicion == tablero[i]+(fila-i)) return false;
        if (tablero[i]==0) return false;
    }
    return true;
}
```

Llega el momento de pensar en cómo vamos a imprimir las soluciones. Estamos hablando de ajedrez, por lo que tendremos que usar la nomenclatura correspondiente: cada posición está definida por la fila (A, B, C…) y la columna (1, 2, 3…). Tendremos que transformar el número de la fila a su letra correspondiente, teniendo en consideración que pueden pedir tableros superiores a 26. Para ello un simple método *String letra(**int** i)* devuelve la letra mayúscula correspondiente al integer pasado.

```
public static String letra(int i) {
    return i < 0 ? "" : letra((i / 26) - 1) + (char)(65 + i % 26);
}
```

El método *ImprimirSolucion()* realiza un recorrido del array de soluciones *tablero[]* imprimiendo cada posición guardada. Si, adicionalmente, se ha activado el modo gráfico, llamará al método

Grafico() para dibujar un tablero después de haber impreso la solución. Se ha añadido un timestamp cada vez que se imprime una solución para calcular el tiempo de ejecución.

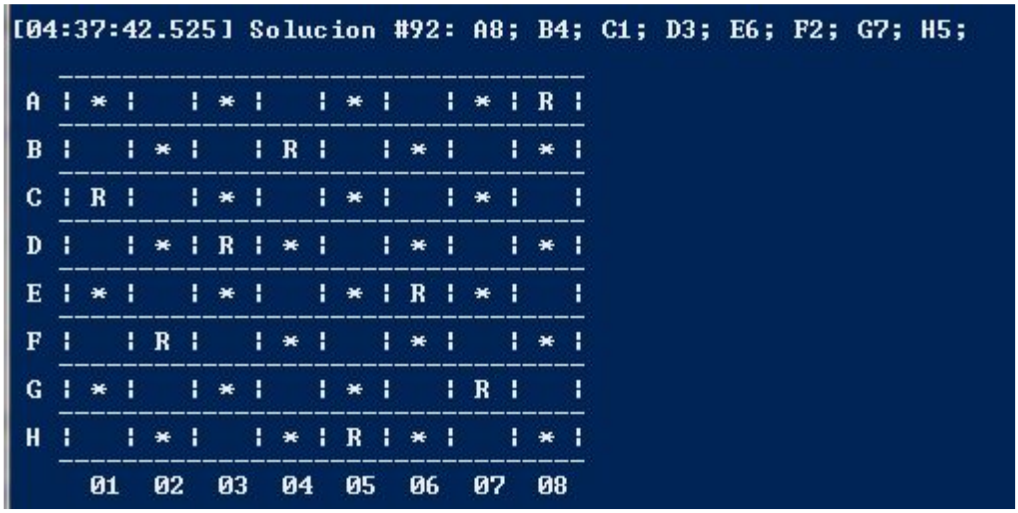
```
public static void ImprimirSolucion() throws IOException {
    SimpleDateFormat time = new SimpleDateFormat("'['hh:mm:ss.SSS']'");
    System.out.print(time.format(new Date())+" Solucion #"+(index+1)+": ");
    for (int i=0; i<n; i++) {
        System.out.print(letra(i)+tablero[i]+"; ");
    }
    System.out.println("\n");
    if (g==true) Grafico();
}
```

```
[04:35:41.068] Solucion #1: A1; B5; C8; D6; E3; F7; G2; H4;
[04:35:41.068] Solucion #2: A1; B6; C8; D3; E7; F4; G2; H5;
[04:35:41.068] Solucion #3: A1; B7; C4; D6; E8; F2; G5; H3;
[04:35:41.078] Solucion #4: A1; B7; C5; D8; E2; F4; G6; H3;
[04:35:41.078] Solucion #5: A2; B4; C6; D8; E3; F1; G7; H5;
[04:35:41.078] Solucion #6: A2; B5; C7; D1; E3; F8; G6; H4;
[04:35:41.078] Solucion #7: A2; B5; C7; D4; E1; F8; G6; H3;
[04:35:41.088] Solucion #8: A2; B6; C1; D7; E4; F8; G3; H5;
```

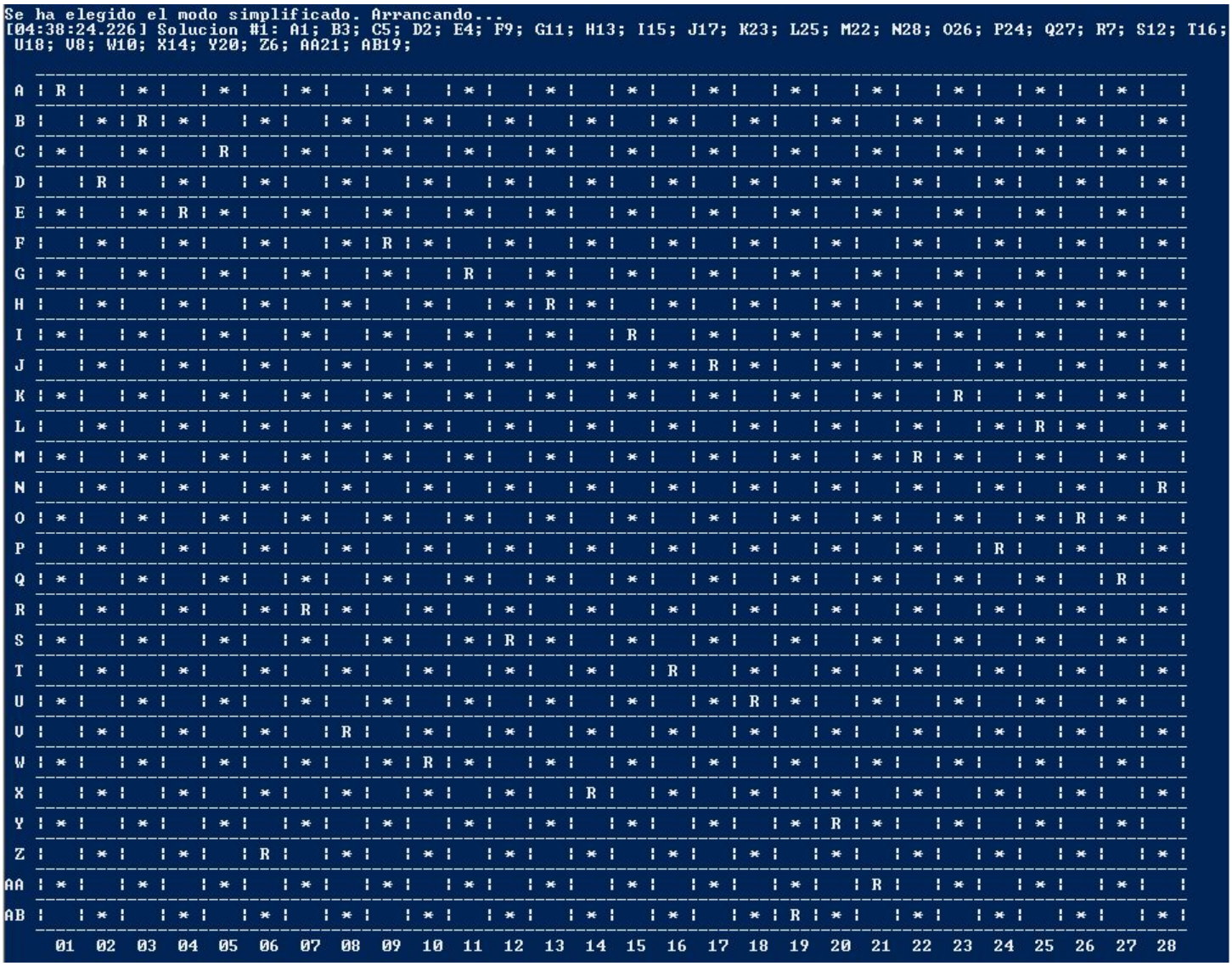
Por último, el método *Grafico()* dibuja el tablero.

```
public static void Grafico() throws IOException {
    String linea = "-";
    String nums ="    ";
    String fila = " |";
    for (int i=0; i<m; i++) {
        linea = linea.concat("----");
        if ((i+1)<10) nums = nums.concat("  0"+(i+1));
        else nums = nums.concat("  "+(i+1));
    }
    for (int i=1; i<=n; i++) {
        System.out.println("   "+linea);
        for (int j=1; j<=m; j++) {
            if (tablero[i-1]==j) fila = fila.concat(" R |");
            else if (i%2==0) {
                if (j%2==0) fila = fila.concat(" * |");
                else fila = fila.concat("   |");
            } else {
                if (j%2==0) fila = fila.concat("   |");
                else fila = fila.concat(" * |");
            }
        }
        if (i<27) System.out.println(" "+letra(i-1)+fila);
        else System.out.println(letra(i-1)+fila);
        fila = " |";
    }
    System.out.println("   "+linea);
    System.out.println(nums+"\n\n");
}
```

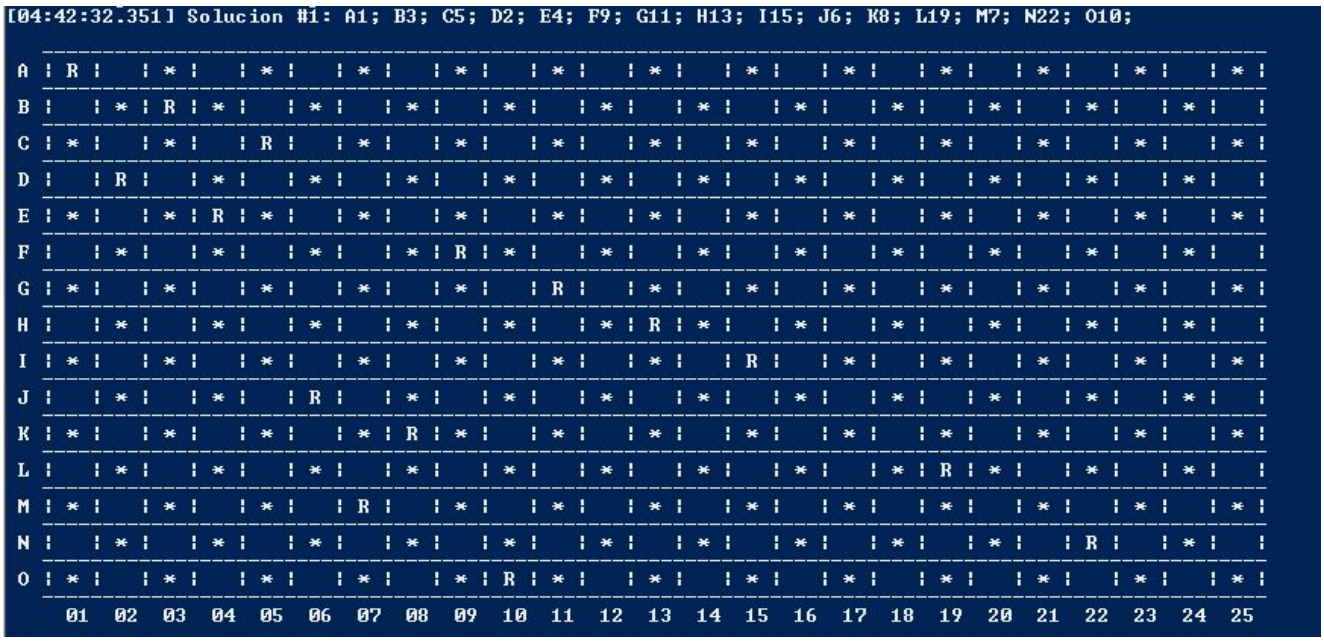
Ejemplo de tablero cuadrado 8x8



Ejemplo de tablero cuadrado de 28x28



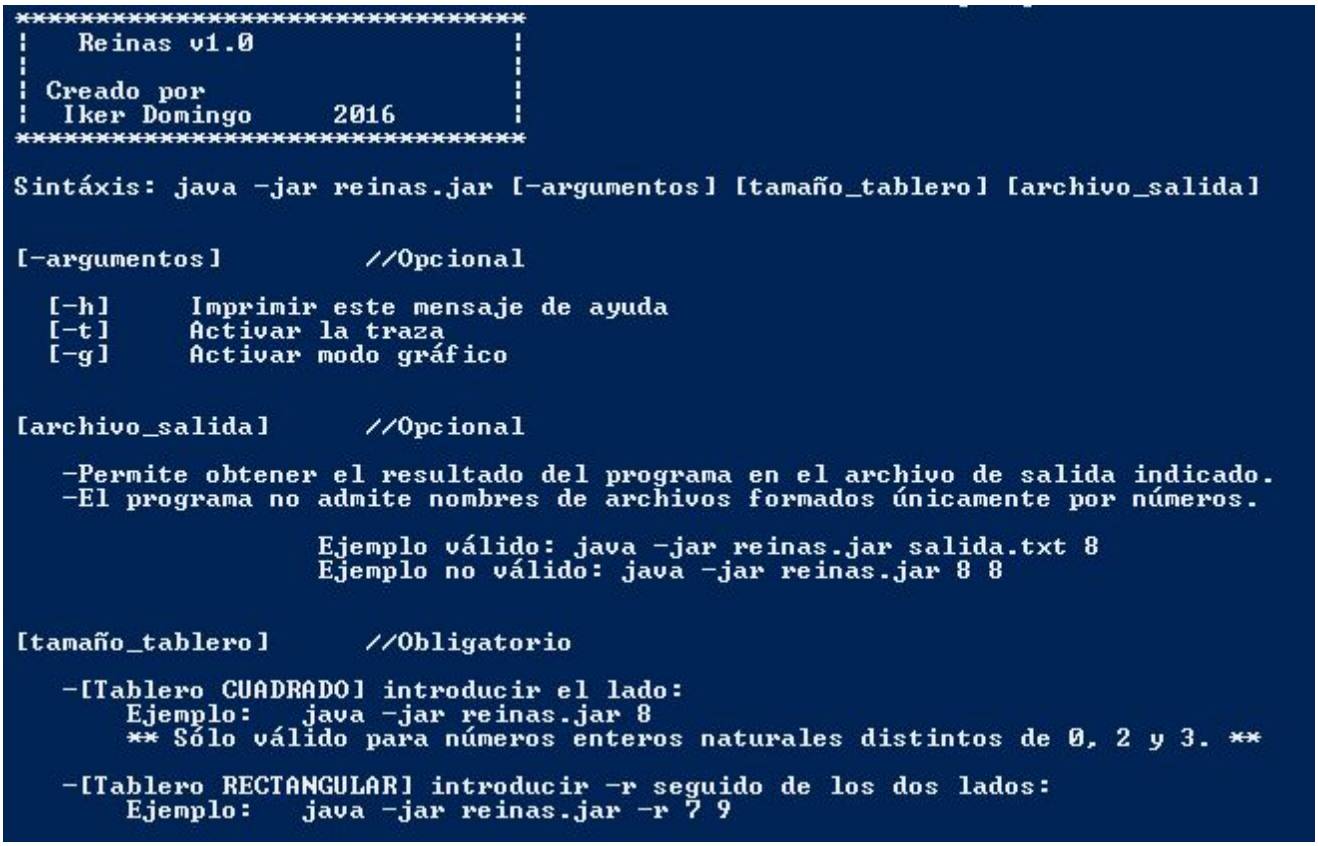
Ejemplo de tablero rectangular 15 x 25



Una última consideración a tener en cuenta es si el usuario ejecuta el programa para un tablero donde uno de los lados sea 1. Si es un cuadrado habrá una única solución, si es un rectángulo habrá tantas soluciones como la longitud del otro lado (***m***). Cuando estemos ante esta situación no hace falta invocar al método *VueltaAtras()*, sino directamente imprimir las soluciones. Esto se realiza en el método *SolucionRapida()*.

```
public static void SolucionRapida() throws IOException {
    for (int i=1; i<=m; i++) {
        tablero[0] = i;
        ImprimirSolucion();
        index++; scan++;
    }
    System.exit(0);
}
```

El siguiente punto a analizar es el análisis de los argumentos. He intentado capturar todas las posibles situaciones susceptibles de producir un error teniendo en cuenta las especificaciones del enunciado de la práctica. Para ello he añadido únicamente un argumento adicional: *-r* para indicar que se va a pedir un tablero rectangular en lugar de cuadrado. El funcionamiento está explicado en el mensaje de ayuda:



Como el análisis de los argumentos es la parte más “liosa” del programa, lo vamos a analizar por partes. Se inicia un bucle *for* que va a recorrer todos los argumentos:

```
for (int i=0; i<args.length; i++) {
```

Ayuda: Imprime el mensaje de ayuda si se pasa ‘-h’, ‘-H’ o ningún argumento

```
if (args[i].toLowerCase().equals("-h")||args.length==0) Error("ayuda", "", "");
```

*** Error(String txt, String err1, String err2) es un método creado para facilitar la lectura del código. Agrupa los distintos mensajes de error que puede lanzar el programa. ***

Traza: Activa la traza cuando se pasa *-t* o *-T*. Cambia el boolean ***t*** a ***true***, crea la carpeta para almacenar los logs si no existiese, y crea el archivo de log con timestamp al milisegundo para evitar problemas con archivos duplicados.

```
else if (args[i].toLowerCase().equals("-t")) t = true;
...
if (t==true) {
    new File("Reinas_logs").mkdir();
    String d = new SimpleDateFormat("'Reinas_Log_'yyyyMMddhhmmssSSS'.log'").format(new Date());
    File f = new File("Reinas_logs", d);
    log = new FileWriter(f);
}
```

De la misma forma que los mensajes de error, los posibles mensajes que se escriben en el log están agrupados en el método *Log(String txt)*.

Grafico: Activa el modo gráfico si se pasa *-g* o *-G*

```
else if (args[i].toLowerCase().equals("-g")) g = true;
```

Rectángulo: Antes de comprobar si el argumento es un número y tomarlo como el lado de un tablero cuadrado, se comprueba si se ha pasado *-r* para indicar que vamos a tratar con un rectángulo. Como se explica en el mensaje de ayuda se debe pasar *-r* seguido de los dos lados, por ejemplo *-r 7 9.* Se revisa que las variables ***n*** y ***m*** no tengan un valor distinto del valor inicial 0 (lo que indicaría que se ha pasado previamente otro valor del lado) y se inicializa el array de soluciones ***int****[m] tablero* poniendo un 0 en todas sus posiciones. Por último se incrementa en dos unidades el índice del bucle *for*, para que en la siguiente iteración se salte los dos números que se esperan leer después de*-r*.

El bloque *try-catch* captura la posible excepción de *Integer.parseInt* por si no se ha utilizado el argumento correctamente.

```
else if (args[i].toLowerCase().equals("-r")) {
    try {
        if (n!=m) Error("duplicado_1", "", "");
        n = Math.min(Integer.parseInt(args[i+1]), Integer.parseInt(args[i+2]));
        m = Math.max(Integer.parseInt(args[i+1]), Integer.parseInt(args[i+2]));
        if (n<=0 || m<=0) Error("rectangulo_1", args[i+1], args[i+2]);
        if (n==m && (n==2 || n==3)) Error("size_2", args[i+1], "");
        tablero = new int[m];
        Arrays.fill(tablero, 0, m, 0);
        i+=2;
    } catch (Exception e) {
        Error("rectangulo_2", args[i+1], args[i+2]);
    }
}
```

Cuadrado: Si el argumento evaluado comienza con un número se entiende que puede ser o bien el lado del tablero o bien el nombre del archivo de salida. No se permite archivos de salida cuyo nombre esté compuesto únicamente por números, por lo que para diferenciarlos intentamos hacer un *Integer.parseInt* del argumento. Si sale bien, se comprueba que ***n*** no tenga un valor diferente del valor inicial 0, es decir, que no se haya pasado otro valor para el lado previamente. Si falla se tratará como nombre de archivo de salida.

Si tenemos un integer, comprobamos su magnitud, debe ser natural distinto de 0, 2 y 3. Si el lado del tablero es válido entonces se inicializa el array de soluciones y se copia el valor de ***n*** en ***m***, ya que es un cuadrado y ambos lados valen lo mismo.

```
else if (Character.isDigit(args[i].charAt(0))) {
    try {
        int j = Integer.parseInt(args[i]);
        if (n!=0) Error("lado_1", args[i], Integer.toString(n));
        n = j;
        if (n==1 || n>=4) {
            tablero = new int[n];
            Arrays.fill(tablero, 0, n, 0);
            m = n;
        } else if (n<=0) Error("size_1", args[i], "");
        else Error("size_2", args[i], "");
    } catch (Exception e) {
        if (s==true) Error("salida_duplicada", args[i], archivo);
        s = true;
        archivo = args[i];
    }
}
```

Archivo de salida: Si el argumento no encaja en los supuestos anteriores se entiende que es el nombre del archivo de salida. Primero se comprueba que no se haya pasado otro argumento previamente como nombre de archivo (lo indica el boolean *s*). Luego se pone el boolean *s* a ***true*** y se copia el valor del argumento en el *String archivo*. Más adelante se crea el archivo de salida comprobando que no exista otro con el mismo nombre, y se captura la posible excepción que tiene lugar cuando existen caracteres inválidos para nombres de archivo (por ejemplo: ?).

```
else {
        if (s==true) Error("salida_duplicada", args[i], archivo);
        s = true;
        archivo = args[i];
    }
...
if (s==true) {
    try {
        File outfile = new File(archivo);
        if (outfile.exists()) Error("salida_1", archivo, "");
        output = new FileWriter (outfile);
    } catch (Exception e) {
        Error("salida_2", archivo, "");
    }
}
```

Al salir del análisis de los argumentos se comprueba que se ha recibido el lado del tablero.

```
if (n==0) Error("no_lado", "", "");
```

También se comprueba, como mencionamos anteriormente, el tamaño del tablero. Para tableros de grandes dimensiones se permite al usuario elegir el modo de ejecución simplificado, donde se obtiene una única solución válida.

```
if (n>=15) {
    InputStream iss = reinas.class.getResourceAsStream("simplificado");
    Scanner scn = new Scanner (iss, "UTF-8");
    String simple = scn.useDelimiter("\\A").next();
    System.out.println(simple);
    System.out.println("\n Por favor, elija el modo de ejecuci\u00f3n (0, 1 \u00f3 2): ");
    do {
        Scanner in = new Scanner(System.in);
        try {
            modo = in.nextInt();
            if (modo==1) {
                System.out.println("Se ha elegido el modo normal. Arrancando...");
            } else if (modo==2) {
                System.out.println("Se ha elegido el modo simplificado. Arrancando...");
                modoSimple = true;
            } else if (modo==0) System.exit(0);
            else System.out.println("Opción no válida...");
        } catch (Exception e) {
            System.out.println("Opción no válida...");
        }
    } while (modo !=0 && modo!=1 && modo!=2);
}
```

Por último, se ejecuta el método *VueltaAtras()* o *SolucionRapida()* si el tablero tiene de lado 1.

```
if (n==1) SolucionRapida();
else VueltaAtras();
```

2.- Analice las condiciones de poda utilizadas y las consideraciones para optimizar la búsqueda.

Como hemos mencionado en el apartado anterior, la poda la se realiza únicamente cuando encontramos un fallo en la última posición de alguno de los niveles.

Ejemplo, *ComprobarSolucion(fila 3, posición 8)* = ***false*** → poda

Da igual si en ese mismo nivel hemos encontrado previamente un acierto o si hemos obtenido una *SoluciónCompleta()*, si se sigue explorando el nivel para más posibles soluciones y se llega al final sin acierto, se corta esa rama y se vuelve atrás a explorar las *OpcionesPendientes()* de la fila anterior.

En cuanto a la optimización de la búsqueda, llevo cerca de un mes dándole vueltas a esta práctica. He depurado el código todo lo posible, eliminando todas las instrucciones innecesarias y no encuentro ninguna forma de mejorar lo presente cuando se trata de obtener todas las soluciones.

He valorado alternativas como guardar en un array adicional las posiciones que van a crear conflicto en las futuras filas. Por ejemplo, Acierto en la fila 1 posición 1, guardar en un array que la fila 2 posición 1 y posición 2 no van a ser válidas, que en la fila 3 las posiciones 1 y 3 tampoco, etc... para así no tener que evaluar posiciones que sabemos de antemano no van a ser válidas.

El método de la vuelta atrás es similar a intentar hackear una contraseña, fuerza bruta, con la diferencia que en nuestro caso es una variación sin repetición, que conocemos las dimensiones y que podemos utilizar la poda para reducir el coste.

3.- Analice el coste computacional y espacial del algoritmo

El coste de un algoritmo que recorre todas las posibles combinaciones de 8 reinas en un tablero (combinaciones sin repetición de 64 elementos tomados de 8 en 8) es ${}_{64}C_{8}$ = 4.426.165.368.

Si acotamos la búsqueda sabiendo que las reinas no se pueden atacar y, por tanto, tendrá que haber una reina por cada fila, el coste se reduce a 8^8 = 16.777.216

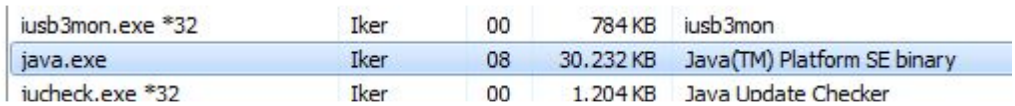
Cuando incluimos el filtro de que las reinas tampoco pueden coincidir en las diagonales, el coste final es de 8! = 40.320.

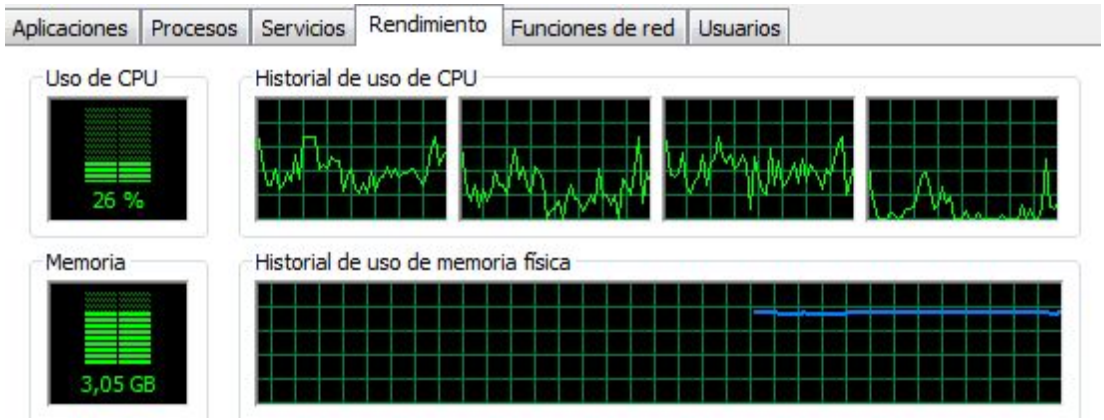
Por tanto, podemos decir que el coste de este algoritmo es de $\Theta(n!)$ siendo ***n*** el tamaño del tablero.

Cuando lo llevamos a la práctica podemos calcular el número de iteraciones del algoritmo, el número de soluciones y el tiempo de ejecución. Hay que tener en cuenta que el tiempo de ejecución se incrementa cuando se activa la traza, cuando se muestra la solución por el terminal y cuando se activa el modo gráfico. En concreto activar la traza aumenta considerablemente el tiempo de ejecución, por ejemplo para un tablero de 14x14, sin traza, se tarda 6 segundos en calcular todas las soluciones. Si se activa la traza tarda en torno a 8 minutos (y genera un archivo log de 5 Gb). El aspecto que tiene el log es el siguiente:

```
[07:18:09.491]Comprobando fila: A; desde posición: 1
[07:18:09.491]  Acierto: A1;
[07:18:09.491]  Acierto: A1; B3;
[07:18:09.491]  Acierto: A1; B3; C5;
[07:18:09.491]  Acierto: A1; B3; C5; D2;
[07:18:09.491]  Acierto: A1; B3; C5; D2; E4;
[07:18:09.491]  Acierto: A1; B3; C5; D2; E4; F9;
[07:18:09.491]Ningún acierto en fila G. -- Vuelta atrás #1 --
[07:18:09.491]Comprobando fila: E; desde posición: 5
[07:18:09.491]  Acierto: A1; B3; C5; D2; E8;
[07:18:09.491]Ningún acierto en fila F. -- Vuelta atrás #2 --
[07:18:09.491]Comprobando fila: E; desde posición: 9
[07:18:09.491]  Acierto: A1; B3; C5; D2; E9;
[07:18:09.491]Ningún acierto en fila F. -- Vuelta atrás #3 --
[07:18:09.491]Comprobando fila: D; desde posición: 3
[07:18:09.491]  Acierto: A1; B3; C5; D7;
[07:18:09.491]  Acierto: A1; B3; C5; D7; E2;
[07:18:09.491]  Acierto: A1; B3; C5; D7; E2; F4;
[07:18:09.491]  Acierto: A1; B3; C5; D7; E2; F4; G6;
[07:18:09.491]Ningún acierto en fila H. -- Vuelta atrás #4 --
[07:18:09.491]Comprobando fila: G; desde posición: 7
```

Durante la ejecución del programa no hemos encontrado problemas donde la aplicación se quede sin memoria o se produzcan cuelgues del sistema. Incluso para tableros de grandes dimensiones podemos observar en la lista de procesos que el consumo de Java está dentro de lo normal, consumiendo unos 30mb de memoria de forma constante y entre el 5% y el 25% de la CPU. Si intentamos asignar más memoria el resultado es el mismo. El tiempo de ejecución del programa no está limitado por la memoria ni por la cantidad de núcleos, sino por la velocidad del procesador.

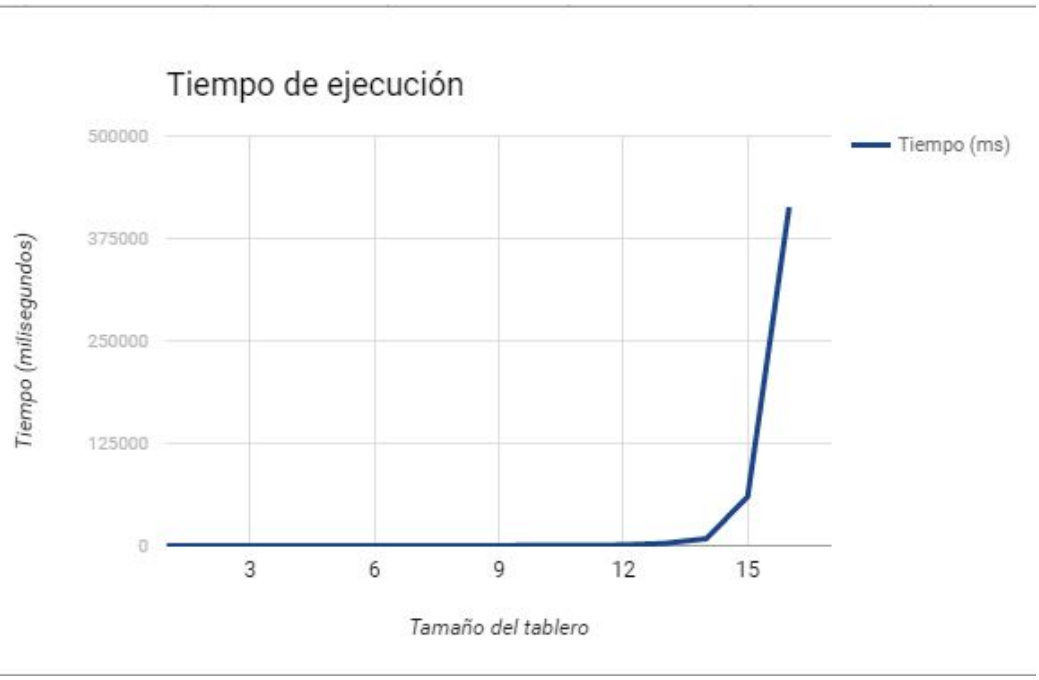




A continuación unas gráficas de la aplicación para tableros de diferentes tamaños.

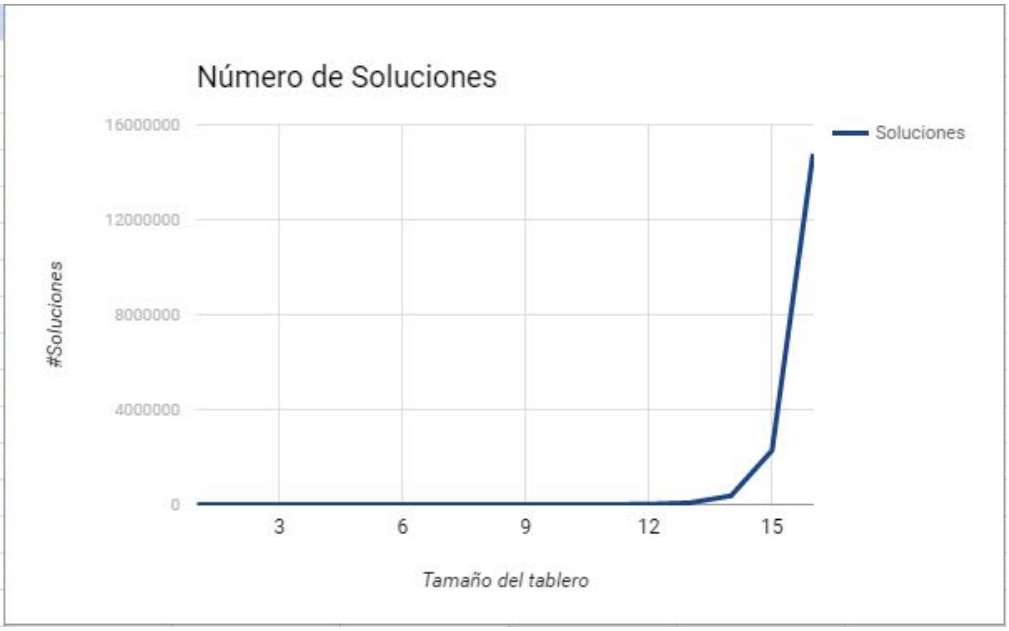
Gráfica 1: Tiempo de ejecución.

Tablero	Tiempo (ms)
1	40
4	40
5	40
6	40
7	60
8	110
9	252
10	294
11	395
12	913
13	2826
14	8544
15	59712
16	413233



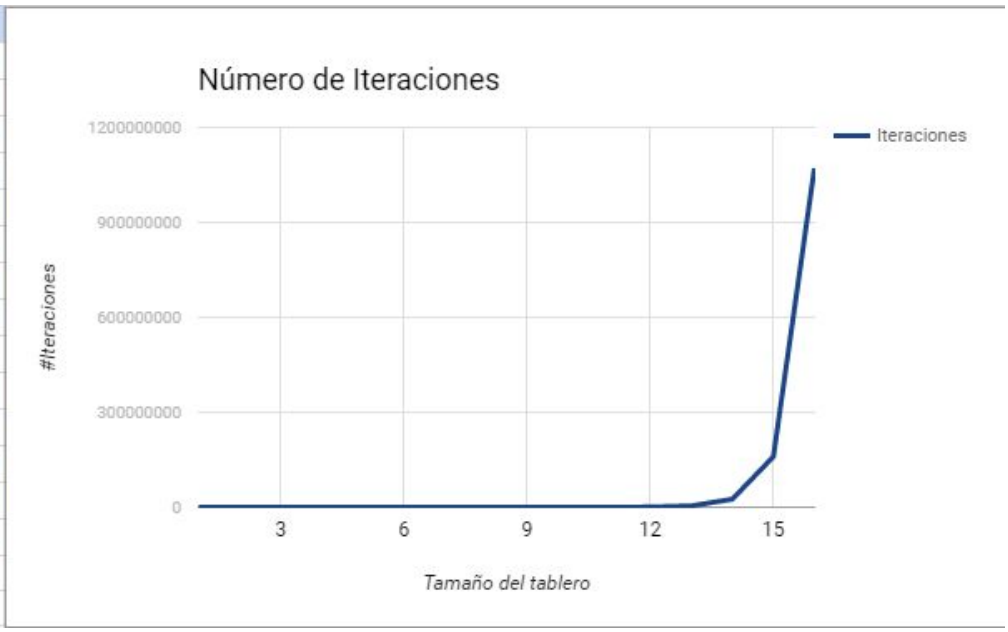
Gráfica 2: Número de soluciones totales.

Tablero	Soluciones
1	1
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512



Gráfica 3: Número de iteraciones del algoritmo.

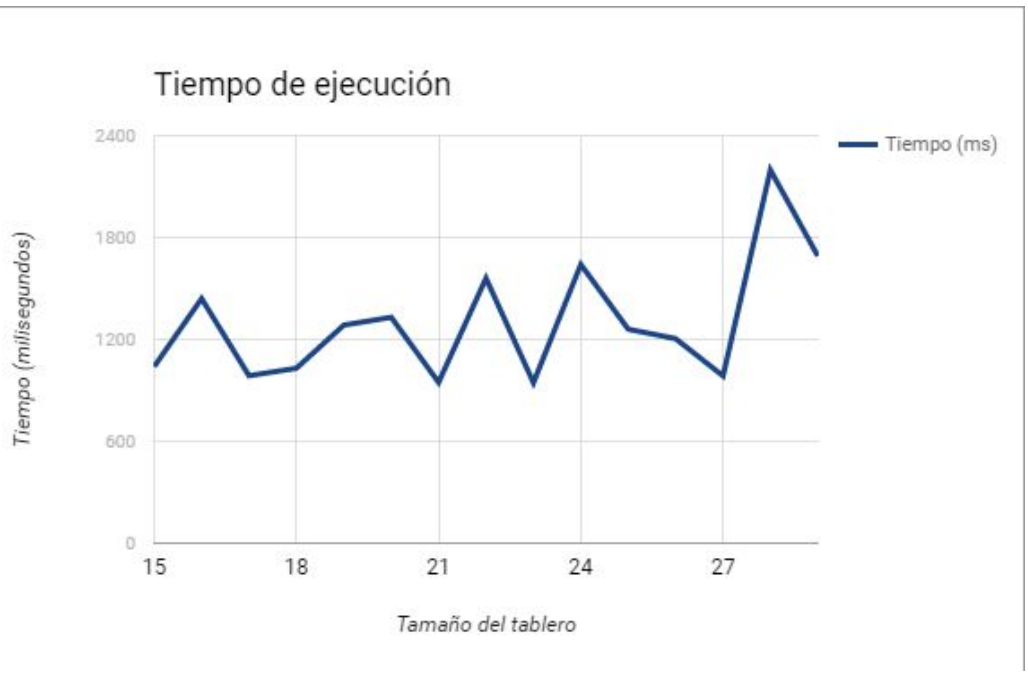
Tablero	Iteraciones
1	1
4	11
5	35
6	127
7	444
8	1742
9	7237
10	31676
11	151093
12	781411
13	4301030
14	25400954
15	159766424
16	1070971322



A la vista está que el incremento es más que significativo a medida que aumenta el tamaño del tablero. También es interesante comparar el tiempo de ejecución y el número de iteraciones que necesita el algoritmo para encontrar una única solución:

Gráfica 4: Tiempo de ejecución para una única solución (15 < n < 29)

Tablero	Tiempo (ms)
15	1040
16	1440
17	986
18	1030
19	1284
20	1330
21	947
22	1560
23	945
24	1641
25	1260
26	1204
27	985
28	2197
29	1691



Gráfica 5: Número de iteraciones para una única solución (15 < n < 29)

Tablero	Iteraciones
15	1200
16	9215
17	4926
18	38445
19	2336
20	189232
21	7980
22	1652530
23	24002
24	392340
25	45972
26	380664
27	429800
28	2876596
29	1464735

A partir de ***n>33*** el tiempo de ejecución y las iteraciones se disparan.

4.- Exponga alternativas al esquema utilizado si las hay, y compare su coste con el de la solución realizada

No he encontrado alternativas al problema de buscar todas las soluciones. Sin embargo, sí hay una alternativa muy interesante para cuando sólo necesitamos una solución válida. Con esta alternativa vamos a reducir el coste de $\Theta(n!)$ a sólo $\Theta(n)$.

Este método, basado en la heurística, consiste en buscar un patrón con forma escalonada y consigue dar con una solución válida de forma directa.

Es válido para cualquier tablero de tamaño ***n*** = *4* o superior y únicamente conoce tres casos distintos:

- Cuando el resto de ***n%6*** = *2*
- Cuando ***n%6*** = *3*
- Cuando ***n%6*** es cualquier otro valor.

En primer lugar cogemos nuestro array de soluciones y, de la posición 0 a *n/2* (la mitad redondeada a la baja) ponemos los números pares (2, 4, 6, 8…).

```
int p = 2;
for (int i=0; i<(n/2); i++) {
    tablero[i] = p;
    p+=2;
}
```

Desde la posición *n/2* hasta el final, los impares (1, 3, 7, 9…) de forma que si ***n*** es impar, habrá un número impar más. (2, 4, 6, 8, 1, 3, 5, 7, 9)

```
p = 1;
for (int i=(n/2); i<n; i++) {
    tablero[i] = p;
    p+=2;
}
```

Cuando *n%6* es distinto de 2 y de 3 ya tenemos nuestra solución.

Cuando el resto de *n%6* = *2,* lo que hacemos es intercambiar las posiciones del 1 y del 3 de la lista de números impares, y a su vez movemos el número 5 al final. (**1**, **3**, **5**, 7, 9 → **3**, **1**, 7, 9, **5**)

```
if (n%6 == 2) {
    int x = tablero[(n/2)];
    tablero[(n/2)] = tablero[(n/2) + 1];
    tablero[(n/2) + 1] = x;
    for (int i = (n/2) + 2; i<(n-1); i++) {
        int y = tablero[i];
        tablero[i] = tablero[i+1];
        tablero[i+1] = y;
    }
}
```

Por último, si el resto de *n%6* = *3*, movemos el 2 al final de los números pares (**2**, 4, 6, 8 → 4, 6, 8, **2**) y también movemos el 1 y el 3 al final de la lista de números impares (**1**, **3**, 5, 7 → 5, 7, **1**, **3**).

```
} else if (n%6 == 3) {
    for (int i = 0; i < (n/2)-1; i++) {
        int x = tablero[i];
        tablero[i] = tablero[i+1];
        tablero[i+1] = x;
    }
    for (int i = 1; i <= 2; i++) {
        for (int j = n/2; j < (n-1); j++) {
            int y = tablero[j];
            tablero[j] = tablero[j+1];
            tablero[j+1] = y;
        }
    }
}
```

Obviamente, con este método no se realiza ninguna vuelta atrás, pero hay que tener en cuenta que el tiempo de ejecución es de unos pocos milisegundos, que ofrece una solución totalmente válida y que también funciona en tableros rectangulares.

En la práctica he dejado este fragmento de código tras comprobar su correcto funcionamiento. La ejecución normal del programa va a seguir realizando la vuelta atrás y sacando todas las posibles soluciones. Este método (llamado *MetodoUltraRapido()*) es llamado cuando, en tableros superiores a 15, el usuario elige la opción de recibir una única solución.