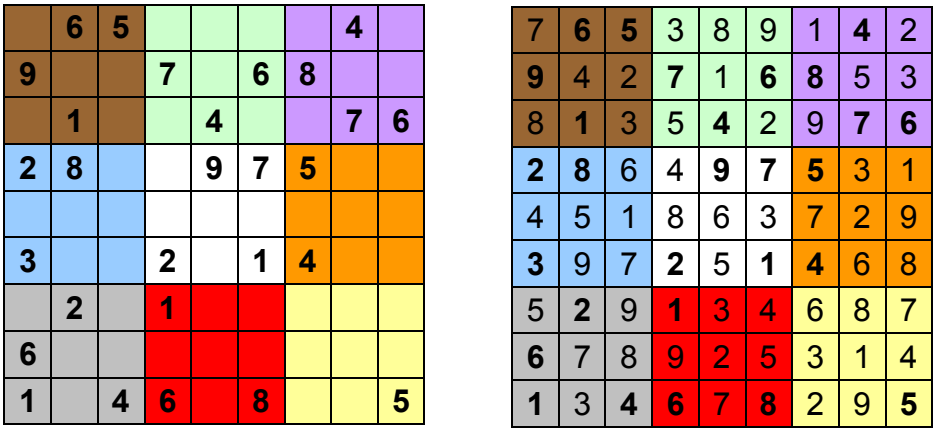
Análisis y Diseño de Algoritmos — Práctica 4

Definición del problema

En un sudoku disponemos de un tablero de tamaño 9x9 en cuyas celdas se pueden situar valores entre 1 y 9. A su vez, el tablero queda dividido en 9 subtableros de tamaño 3x3 (indicados con distintos colores en las figuras posteriores). El valor de algunas celdas está fijado inicialmente. El juego consiste en completar los valores de las demás celdas de modo que se cumplan las siguientes reglas:

1) un mismo valor no puede aparecer más de una vez en la misma fila del tablero,
2) un mismo valor no puede aparecer más de una vez en la misma columna del tablero,
3) y un mismo valor no puede aparecer más de una vez en el mismo subtablero.

Por ejemplo, en la siguiente figura, el tablero a la derecha es una solución válida para la configuración inicial dada en la parte izquierda:



Aunque normalmente se usan configuraciones iniciales que solo permiten una única solución (son lo que se llaman *sudokus bien formados*), en esta práctica consideraremos sudokus que puedan admitir varias soluciones correctas.

Objetivos de la práctica

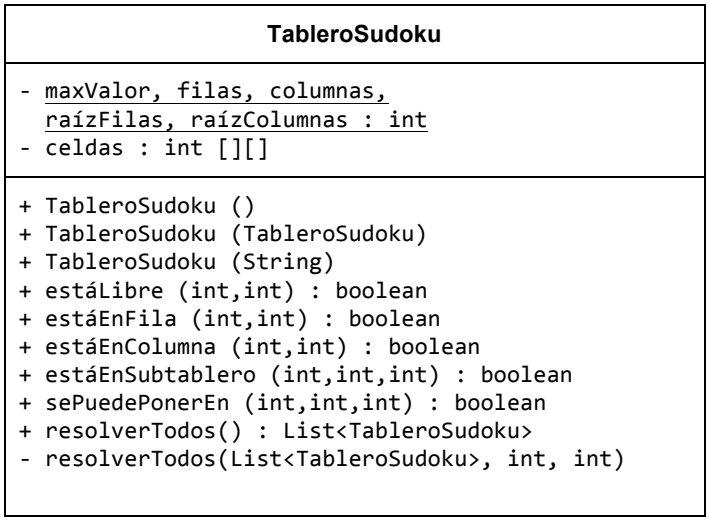
El alumno deberá:

a) Implementar un algoritmo, mediante la técnica de vuelta atrás, que obtenga todas las soluciones posibles para una configuración inicial de sudoku.

b) Evaluar experimentalmente la corrección de su implementación, tal como se explica posteriormente.

Modelado del problema

Para resolver este problema, definiremos una clase `TableroSudoku`, de modo que un objeto instancia de la clase representa una instancia del problema:



Las distintas constantes estáticas de la clase tienen la siguiente interpretación: `maxValor` es 9, el máximo valor que se puede colocar en una celda. `filas` y `columnas` valen 9, y representan respectivamente el nº de filas y columnas del tablero. `raízFilas` y `raízColumnas` valen 3, y representan respectivamente el nº de filas y columnas de un subtablero. La variable de instancia `celdas` es una matriz de enteros que representa el contenido del tablero. Numeraremos las filas y columnas del tablero de 0 a 8. Como convenio, indicaremos que una celda está vacía poniendo un cero en la correspondiente posición de la matriz de enteros `celdas`. Una celda no vacía contendrá un número entre 1 y 9.

Existen constructores (ya definidos en el esqueleto) para crear un tablero con todas sus celdas vacías, para crear un tablero como copia de otro, y para crear un tablero a partir de una cadena de caracteres. Esta cadena de 81 caracteres consistirá en el contenido de cada una de las celdas del tablero al recorrerlo por filas, de izquierda a derecha. Cada uno de los caracteres de la cadena podrá ser un dígito entre 1 y 9, o un **“.”** para representar una celda vacía. Por ejemplo, el tablero en la parte izquierda de la figura anterior se crearía con:

```
new  TableroSudoku (
  “.65….4.9..7.68…1..4..7628..975……..3..2.14…2.1…..6……..1.46.8..5”)
```

El método (ya definido) `estáLibre` toma como parámetros un nº de fila y un nº de columna, y devuelve `true` si dicha celda está vacía.

Los siguientes métodos auxiliares deberán ser implementados por el alumno:

- **`boolean`** `estáEnFila(`**`int`** `fila, `**`int`** `valor)`, que devuelve `true` si el valor `valor` ya aparece en alguna posición de la fila `fila`.
- **`boolean`** `estáEnColumna(`**`int`** `columna, `**`int`** `valor)`, que devuelve `true` si el valor `valor` ya aparece en alguna posición de la columna `columna`.

- **boolean** estáEnSubtablero (**int** fila, **int** columna, **int** valor), que devuelve `true` si el valor `valor` ya aparece en alguna posición dentro del subtablero al que corresponde la posición dada por `fila` y `columna`.
- **boolean** sePuedePonerEn(**int** fila, **int** columna, **int** valor), que devuelve `true` si el valor `valor` puede colocarse en la celda dada por `fila` y `columna`.

La resolución del problema, mediante la técnica de vuelta atrás, se realizará completando los métodos `resolverTodos`. Se proporciona la implementación de la versión sin parámetros, que es la siguiente:

```
public List<TableroSudoku> resolverTodos() {
 List<TableroSudoku> sols = new LinkedList<TableroSudoku>();
 resolverTodos(sols, 0, 0);
      return sols;
}
```

Éste será el método que finalmente habrá que invocar para obtener una lista de tableros con cada una de las soluciones a la instancia. El trabajo del alumno es implementar el método recursivo auxiliar:

```
private void resolverTodos ( List<TableroSudoku> soluciones
                           , int fila, int columna)
```

que añada a la lista `soluciones` todas las soluciones válidas para el tablero que se pueden obtener rellenando las celdas a partir de la fila `fila` y la columna `columna` inclusive. Al invocar este método con una lista vacía y partiendo de la celda inicial del tablero (tal como se hace en la versión sin parámetros del método), se obtendrán en la lista todos los tableros solución de la instancia.

Básicamente, la resolución con vuelta atrás consistirá en colocar uno de los valores válidos en la celda actual (si ésta está libre) e intentar completar el resto del tablero. En caso de que no sea posible completar el resto del tablero con el valor colocado, habrá que repetir la operación con otro valor. Cada vez que se consiga colocar la última celda del tablero, tendremos una solución completa, la cual habrá que añadir a la lista de soluciones.

La interfaz List<TableroSudoku>

La interfaz paramétrica `List<TableroSudoku>` del paquete `java.util` representa listas que almacenan valores de la clase `TableroSudoku`. Por ejemplo, podemos crear una lista con dos tableros del siguiente modo:

```
List<TableroSudoku> ls = new LinkedList<TableroSudoku>();
TableroSudoku t1 = new TableroSudoku(
  ".....64.9.5..9.....4183....9..7...637..2..1.2.8..9.7.....862.....1.45.....1..");

ls.add(t1);
ls.add(new TableroSudoku());
```

NOTA: Observa que, en el método `resolverTodos(List<TableroSudoku> soluciones, int fila, int columna)`, habrá que añadir a la lista `soluciones` el tablero actual cada vez que está completo. Si intentamos esto del siguiente modo:

```
soluciones.add(this);
```

introduciremos en la lista el propio objeto que contiene actualmente la solución, pero si modificamos éste posteriormente (cuando estemos buscando nuevas soluciones), modificaremos también el introducido en la lista (es decir, no se introduce una copia del objeto `this` en la lista, sino el propio objeto). Lo correcto en este caso es introducir una copia del objeto `this` en la lista cada vez que se encuentre una solución, lo cual puede hacerse así:

```
soluciones.add(new TableroSudoku(this));
```

que usa uno de los constructores proporcionados para hacer una copia del valor actual.

Evaluación experimental de la implementación

Con objeto de que el alumno compruebe experimentalmente el funcionamiento de su implementación, se suministra la clase `TestsCorrección`, que comprueba que la implementación es correcta para un conjunto de instancias almacenadas en el fichero "`tests.txt`". **Es obligatorio que la implementación del alumno supere la prueba que se realiza al ejecutar el método `main` de esta clase**. Dicha prueba comprueba el funcionamiento del método `resolverTodos()`. Para una implementación correcta, la salida por pantalla tras ejecutar este método debe ser:

```
Test de resolverTodos correcto
```

También hay un ejemplo mas pequeño en el método `main` de la clase `TableroSudoku.java`

Entrega

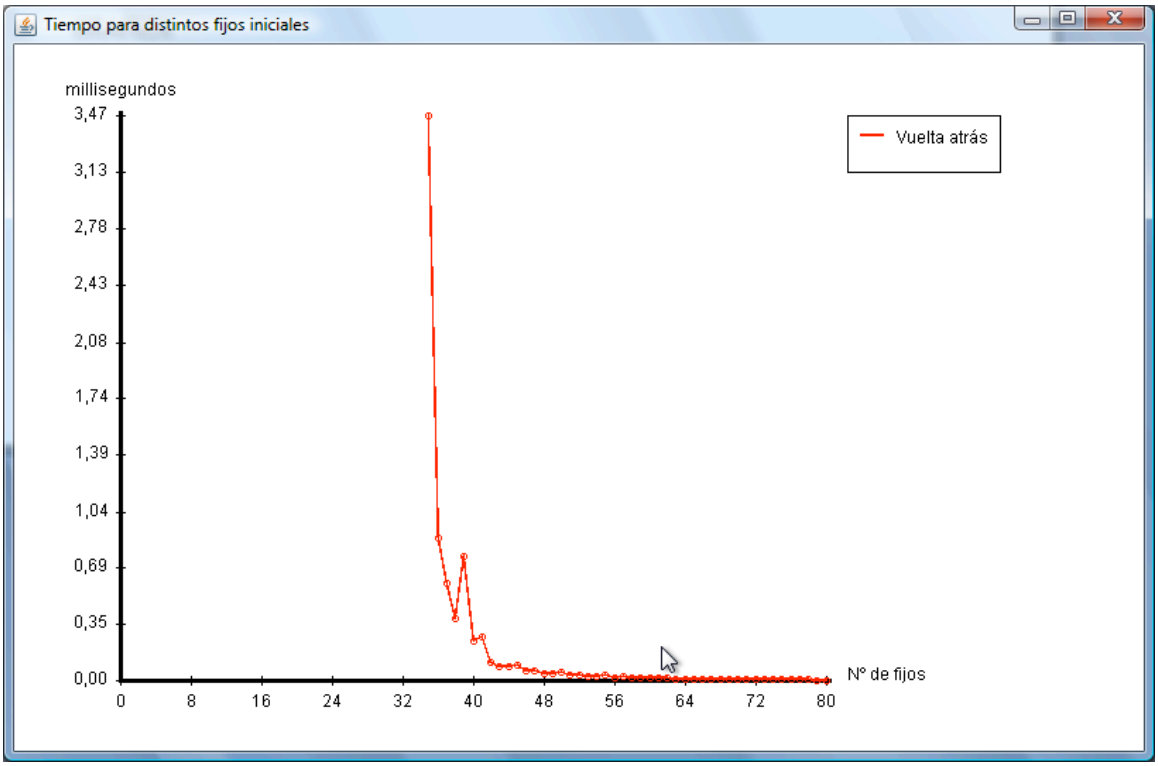
Una vez finalizada la practica, debe entregarse el programa realizado para su corrección de forma automática mediante el programa *Siette*. Para ello, seleccionar la actividad correspondiente en el Campus Virtual y enviar, una vez modificado, el fichero:

`TableroSudoku.java`

La corrección se realiza con otros sudokus generados al azar, y consiste en diversas pruebas que verifican que la solución cumple diversas condiciones, por ejemplo que se han encontrado todas las soluciones posibles, que las soluciones cumplen todas o algunas de las restricciones, etc. Se pueden obtener puntuaciones parciales en caso de que supere alguna de las pruebas, por ejemplo si encuentra al menos una solución.

La clase Gráfica

Con la clases `Temporizador` y `Gráfica` suministradas, podrás realizar gráficas para ver experimentalmente el comportamiento de tu implementación, como la siguiente, que se obtuvo en un Pentium Core2 6300 y muestran el tiempo medio para resolver sudokus con distinto número de celdas fijadas inicialmente:



Para ver el comportamiento experimental de tu implementación, ejecuta la clase `TestsTiempos`.