PED 2: Sudoku

Programación y Estructuras de Datos Avanzadas

ADRIANA ARMENTAL TOMÉ

53489431-X

adri.13a@gmail.com

Índice

Enunciado de la practica: Sudoku

El juego del Sudoku consiste en rellenar un cubo de 9 x 9 celdas dispuestas en 9 subgrupos de 3 x 3 celdas, con números del 1 al 9, atendiendo a la restricción de que no se debe repetir el mismo número en la misma fila, columna o subgrupo de 9. Un Sudoku dispone de varias celdas con un valor inicial, de modo que debemos empezar a resolver el problema a partir de esta solución parcial sin modificar ninguna de las celdas iniciales.

La práctica constará de un programa en java que resuelva el problema aplicando el esquema de Vuelta Atrás (Backtracking) junto con una memoria de su implementación.

Esquema algorítmico utilizado y otros que lo puedan resolver

Existen problemas para los que no parece existir una forma o regla fija que nos lleve a obtener una solución de una manera eficiente y precisa. La manera de resolverlos consiste en realizar una búsqueda exhaustiva entre todas las soluciones potenciales hasta encontrar una solución válida, o el conjunto de todas las soluciones válidas. A veces se requiere encontrar la mejor (en un sentido preciso) de todas las soluciones válidas.

La búsqueda exhaustiva en un espacio finito dado se conoce como fuerza bruta y consiste en ir probando sistemáticamente todas las soluciones potenciales hasta encontrar una solución satisfactoria, o bien agotar el universo de posibilidades. En general, la fuerza bruta es impracticable para espacios de soluciones potenciales grandes, lo cual ocurre muy a menudo, ya que el número de soluciones potenciales tiende a crecer de forma exponencial con respecto al tamaño de la entrada en la mayoría de los problemas que trataremos.

El esquema algorítmico de vuelta atrás es una mejora a la estrategia de fuerza bruta, ya que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir el espacio de búsqueda. La diferencia principal entre ambos esquemas es que en el primero las soluciones se forman de manera progresiva, generando soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria. Mientras que en la fuerza bruta la estrategia consiste en probar una solución potencial completa tras otra sin ningún criterio.

En el esquema de vuelta atrás, si una solución parcial no puede llevar a una solución completa satisfactoria, la búsqueda se aborta y se vuelve a una solución parcial viable, deshaciendo decisiones previas. Este esquema debe su nombre a este salto hacia atrás.

Tomemos como ejemplo el siguiente problema: dadas n letras diferentes, diseñar un algoritmo que calcule las palabras con m letras (m ð n) diferentes escogidas entre las dadas. El orden de las letras es importante: no será la misma solución abc que bac. El número de soluciones potenciales o espacio de búsqueda es de nm, que representan las variaciones con repetición de n letras tomadas de m en m. Realizar una búsqueda exhaustiva en este espacio es impracticable.

Antes de ponernos a probar sin criterio alguno entre todas las posibles combinaciones de letras, dediquemos un segundo a evaluar la estructura de la solución. Es evidente que no podemos poner dos letras iguales en la misma palabra, así que vamos a replantear el problema. Trataremos de colocar las letras de una en una, de forma que no se repitan. De esta manera, toda solución del problema se puede representar como una tupla (x_1, ..,x_m) en la que xi representa la letra que se coloca en el lugar i-ésimo de la palabra.

La solución del problema se construye de manera incremental, colocando una letra detrás de otra. En cada paso se comprueba que la última letra no esté repetida con las anteriores. Si la última letra colocada no está repetida, la solución parcial se dice prometedora y la búsqueda de la solución continúa a partir de ella. Si no es prometedora, se abortan todas las búsquedas que partan de esa tupla parcial.

De manera general, en los algoritmos de vuelta atrás, se consideran problemas cuyas soluciones se puedan construir por etapas. Una solución se expresa como n-tupla (x_1, ..,x_n) donde cada x_i € S_i representa la decisión tomada en la i-ésima etapa de entre un conjunto finito de alternativas.

Una solución tendrá que minimizar, maximizar, o simplemente satisfacer cierta función criterio. Se establecen dos categorías de restricciones para los posibles valores de una tupla:

- **Restricciones explícitas**, que indican los conjuntos Si. Es decir, el conjunto finito de alternativas entre las cuales pueden tomar valor cada una de las componentes de la tupla solución.
- **Restricciones implícitas**, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Puesto en forma de código, el esquema de Vuelta atrás tiene el siguiente aspecto:

```
vueltaAtras (Tupla & sol, int k) {
    prepararRecorridoNivel(k);
    while(!ultimoHijoNivel(k)){
        sol[k] = siguienteHijoNivel(k);
        if (esValida(sol, k)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else
                vueltaAtras(sol, k + 1);
        }
    }
}
```

El tipo de la solución sol es una tupla de cierto tipo específico para cada problema. En ella se va acumulando la solución. La variable k es la que determina en qué nivel del árbol de exploración estamos.

El método prepararRecorridoNivel genera los candidatos para ampliar la solución en la siguiente etapa y depende del problema en concreto. En el cuerpo de la función, iteramos a lo largo de todas las posibles soluciones candidatas, dadas por la función: siguienteHijoNivel hasta la última candidata, dada por la función: ultimoHijoNivel. Para cada solución candidata, ampliamos la solución con el nuevo valor y comprobamos si satisface las restricciones implícitas/explícitas con la función booleana esValida. Esta función implementa la función de factibilidad que presentábamos más arriba.

En el caso de que la solución parcial sea válida tenemos dos posibilidades: o bien hemos alcanzado el final de la búsqueda, por lo que ya podemos mostrar la solución final, o bien continuamos nuestra búsqueda mediante la llamada recursiva.

Este esquema encontrará todas las soluciones del problema. Si quisiéramos que sólo encontrara una solución bastaría con añadir una variable booleana éxito que haga finalizar los bucles cuando se encuentra la primera solución.

El árbol de exploración generado tendrá las siguientes características:

- Altura = m + 1: Siendo m el número de casillas vacías inicialmente.
- Nº de Hijos de cada nodo = n: Un hijo por cada posible valor de la celda i j.

Proc sudoku_VA (i, j: Nat; sol[1..n, 1..n] de 0..n; inicial[1..n, 1..n] de Bool)

```
   Si (inicial [i, j] = Falso) Entonces
      Para (k := 1) Hasta n Hacer
         sol[i, j] := k;                                    //marcar
         Si (es_factible (i, j, sol)) Entonces
            Casos
               i = n ^ j = n -> mostrarPorPantalla(sol);
               i < n ^ j = n -> sudoku_VA (i+1, 1, sol, inicial);
               i <= n ^ j < n -> sudoku_VA( i , j+1, sol, inicial);
            FinCasos;
         FinSi;
         sol[i, j] : = 0;
//Desmarcar
      FinPara;
   En Otro Caso //inicial[i, j] = Cierto
      Casos
         i = n ^ j = n -> mostrarPorPantalla(sol);
         i < n ^ j = n -> sudoku_VA (i+1, 1, sol, inicial);
         i <= n ^ j < n -> sudoku_VA( i , j+1, sol, inicial);
      FinCasos;
   FinSi;
FinProc;

Proc sudoku (sol[1..n, 1..n] de 0..n)

   Var
      inicial[1..n, 1..n] de Bool;
   FinVar;
   Para (i := 1) Hasta n Hacer
      Para (j := 1) Hasta n Hacer
            inicial[i, j] := Sol[i, j] != 0;
      FinPara;
   FinPara;
   sudoku_VA(1, 1, sol, inicial);
FinProc;

Fun es_factible (i, j : Nat; sol[1..n, 1..n] de 0..n) DEV Bool
```

```
   Var
      valido : Bool;
      k,l: Nat;
   FinVar;
   valido := True;
   k := 1;
   Mientras (k <= n ^ valido) Hacer                    //Comprobamos
la columna
      Si ( sol[i, j] = sol[k, j] ^ k != i ){
         Valido := Falso;
      FinSi;
      k := k + 1;
   FinMientras;
   l := 1;
   Mientras (l <= n ^ valido) Hacer                    //Comprobamos
la fila
      Si ( sol[i, j] = sol[i, l] ^ l != j ){
         Valido := Falso;
      FinSi;
      l := l + 1;
   FinMientras;
   //                          Lo anterior podría compactarse así,
en un solo while que comprueba filas y columnas..
   // Mientras (k<=n ^ valido) Hacer
   //    Si ( (sol[i, j] = sol[k, j] ^ k != i) v (sol[i, j] =
sol[i, k] ^ k != j))
   //       Valido := Falso;
   //    FinSi;
   // FinMientras;
   k := correspondencianxn(i);
   l :=  correspondencianxn(j);
//Comprobamos el subgrupo de nxn
   Mientras ( k < correspondencianxn(i) + sqrt{n} ^ valido )
Hacer //por razones de eficiencia puede antes de esta etapa,
asignar a una variable
      Mientras ( l < correspondencia3x3(j) + sqrt{n} ^ valido)
Hacer // el valor de correspondencia3x3(x) sea x=i o = j; así se
evitan 2 llamadas
         Si ( sol[i, j] = sol[k, l] ^ i != k ^ j != l) Entonces //
a dicha función traduciéndose en mejor eficiencia.
            valido := Falso;
         FinSi;
         l := l + 1;
      FinMientras;
      k := k + 1;
```

```
      l :=  correspondencianxn(j);
   FinMientras;
   Devolver valido;
FinFun;

Fun correspondencianxn (i: Nat) DEV Nat

   Var
      k : Nat;
      resultado: Nat;
   FinVar;
   Si ( i MOD sqrt{n} = 0) Entonces
      k := (i DIV sqrt{n});
   En Otro Caso
      k := ( I DIV sqrt{n}) + 1;
   FinSi;
   Casos
      k = 1 -> resultado := 1;
      k = 2 -> resultado := 4;
      k = 3 -> resultado := 7;
   FinCasos;
   Devolver resultado;
FinFun;
```

Complejidad del tiempo: O (n ^ (n * n)).
Dependiendo del grado del sudoku para cada índice no asignado, hay n opciones posibles, por lo que la complejidad del tiempo es O (n ^ (n * n)). La complejidad del tiempo sigue siendo la misma, pero habrá una poda temprana, por lo que el tiempo necesario será mucho menor que el del algoritmo ingenuo, pero la complejidad del límite superior seguirá siendo la misma.

Complejidad espacial: O (n * n).
Para almacenar el array de salida se necesita una matriz.

También se puede resolver por ramificación y poda como comentaré en el apartado siguiente.

Heurísticas que permiten condiciones de poda

El juego del Sudoku no es un problema de optimización, con lo cual no recorreremos el árbol de búsqueda guiándonos con una función de coste.

A diferencia de los algoritmos de Vuelta Atrás, con Ramificación y Poda podemos hacer un recorrido por niveles del árbol de exploración, gestionando los nodos vivos con una cola.

El tablero del Sudoku a resolver viene dado por una matriz “Sol [1..n,1..n] de 0..n” donde Sol[i, j] representa el valor que toma dicha celda, correspondiéndose el valor 0 con una casilla vacía.

Se utilizará una matriz auxiliar “inicial[1..n, 1..n] de Bool” donde inicial[i, j] representa una celda con valor inicial que no se puede modificar y se corresponde con la celda “Sol[i, j]”.

A la hora de ramificar el árbol de exploración, solo lo haremos si la solución parcial que estamos atendiendo es k-prometedora, esto es, si a partir de dicha solución parcial podremos seguir construyendo soluciones parciales. Para atender a este punto, utilizaremos una función auxiliar denominada “es_factible”.

La función “es_factible” comprueba para una celda determinada, que no se repita su valor en la misma fila, columna o subgrupo de $\sqrt{n}$ x $\sqrt{n}$, atendiendo así a la restricción que comentábamos en la descripción detallada del problema.

Dado que un Sudoku puede tener varias soluciones, implementaremos el algoritmo en consecuencia.

El árbol de exploración generado tendrá las siguientes características:

- Altura = m + 1, siendo m el número de casillas vacías inicialmente.
- Nº máximo de Hijos de cada nodo = n, un hijo por cada posible valor de la celda i j.

```
Fun sudoku_RyP ( sol[1..n, 1..n] de 0..n)
   Var
      Vivos: cola de NODO;
      X,Y: NODO;
      inicial[1..n, 1..n] de Bool;
   FVar;
   Vivos := cola_vacia();              //Preparamos la raíz del
árbol de exploración
   X.fila := 1;
   X.columna := 1;
   X.sol := sol;
   pedir_vez (Vivos, X);
   Para (i := 1) Hasta n Hacer         //Inicialización de la
matriz de iniciales
      Para (j := 1) Hasta n Hacer
         Si (Sol[i, j] != 0) Entonces
```

```
            Inicial[i, j] := Falso;
         En Otro Caso
            Inicial[i, j] := Cierto;
         FSi;
      FPara;
   FPara;
   Mientras ( cola_vacia(Vivos) = Falso ) Hacer
      X := atender (Vivos);
      Si (inicial [X.fila, X.columna] = Falso) Entonces
         Para (k := 1) Hasta n Hacer
            X.sol[X.fila, X.columna] := k;
            Si (es_factible (X.fila, X.columna, X.sol)) Entonces
               Casos
                  X.fila = n ^ X.columna = n ->
mostrarPorPantalla( X.sol);
                  X.fila < n ^ X.columna = n -> Y.sol := X.sol;
                                                Y.fila := X.fila + 1;
                                                Y.columna := 1;
                                                pedir_vez(Vivos, Y);
                  X.fila <= n ^ X.columna < n -> Y.sol := X.sol;
                                                 Y.fila := X.fila;
                                                 Y.columna :=
Y.columna + 1;
                                                 pedir_vez(Vivos, Y);
               FCasos;
            FSi;
         FPara;
      En Otro Caso //inicial[X.fila, X.columna] = Cierto
         Casos
            X.fila = n ^ X.columna = n ->
mostrarPorPantalla( X.sol);
            X.fila < n ^ X.columna = n -> Y.sol := X.sol;
                                          Y.fila := X.fila + 1;
                                          Y.columna := 1;
                                          pedir_vez(Vivos, Y);
            X.fila <= n ^ X.columna < n -> Y.sol := X.sol;
                                           Y.fila := X.fila;
                                           Y.columna := Y.columna +
1;
                                           pedir_vez(Vivos, Y);
         FCasos;
      FSi;
   FMientras;
FFun;
```

```
Fun es_factible (i, j : Nat; sol[1..n, 1..n] de 0..n) DEV Bool
   Var
      valido : Bool;
      k, l : Nat;
   FVar;
   valido := Cierto;
   k := 1;
   Mientras (k <= n ^ valido) Hacer                    //Comprobamos
la fila
      Si ( sol[i, j] = sol[i, k] ^  k != j ) Hacer
         valido := Falso;
      FSi;
   FMientras;
   k := 1;
   Mientras (k <= n ^ valido) Hacer                     //Comprobamos
la columna
      Si ( sol[i, j] = sol[k, j] ^ k != i ){
         Valido := Falso;
      FSi;
   FMientras;
   k := correspondencianxn(i);
   l :=  correspondencianxn(j);
//Comprobamos el subgrupo de nxn
   Mientras ( k < correspondencianxn(i) +  sqrt{n} ^ valido ) Hacer
      Mientras ( l < correspondencianxn(j) + sqrt{n} ^ valido)
Hacer
         Si ( sol[i, j] = sol[k, l] ^ i != k ^ j != l) Entonces
            valido := Falso;
         FSi;
      FMientras;
   FMientras;
   Devolver valido;
FFun;
```

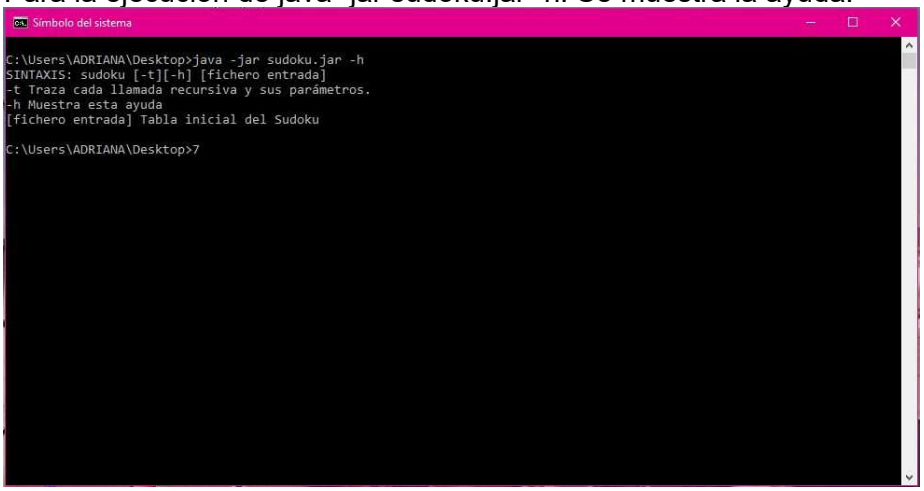
```
Fun correspondencianxn (i: Nat) DEV Nat
   Var
      k : Nat;
      resultado: Nat;
   FVar;
   Si ( i MOD sqrt{n} = 0) Entonces
      k := (i DIV sqrt{n});
   En Otro Caso
      k := ( i DIV sqrt{n}) + 1;
   FSi;
   Casos
```

```
      k = 1 -> resultado := 1;
      k = 2 -> resultado := 4;
      k = 3 -> resultado := 7;
   FCasos;
   Devolver resultado;
FFun;
```

Ejemplos de ejecución

El programa se ejecuta usando la secuencia java -jar.

Para la ejecución de java -jar sudoku.jar -h. Se muestra la ayuda.



Para la ejecución de java -jar sudoku.jar (por defecto tiene un sudoku de 9x9).

Para la ejecución del programa leyendo el archivo, tiene que ser un .txt de la siguiente forma (solo se permite este formato).

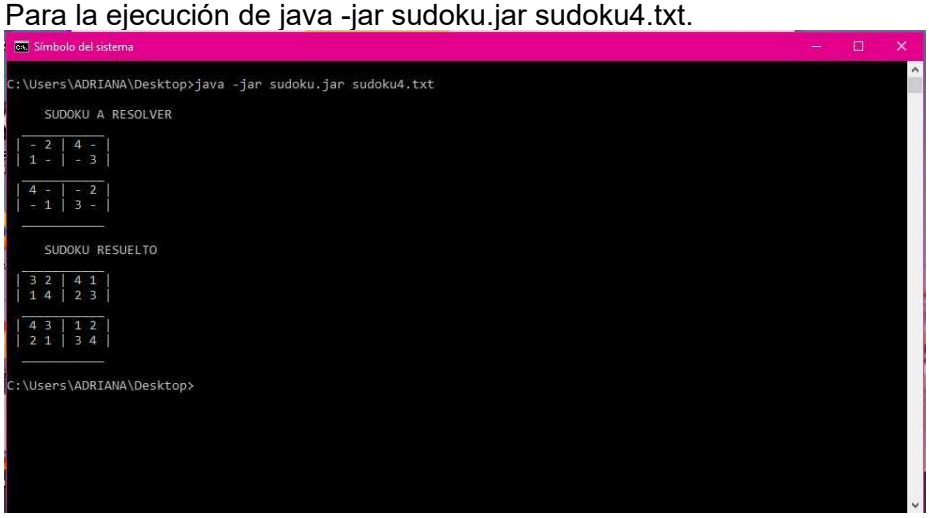
```
8 - -  - - -  - - -
- - 3  6 - -  - - -
- 7 -  - 9 -  2 - -

- 5 -  - - 7  - - -
- - -  - 4 5  7 - 1
- - -  1 - -  - 3 -

- - 1  - - -  - 6 8
- - 8  5 - -  - 1 -
- 9 -  - - -  4 - -
```

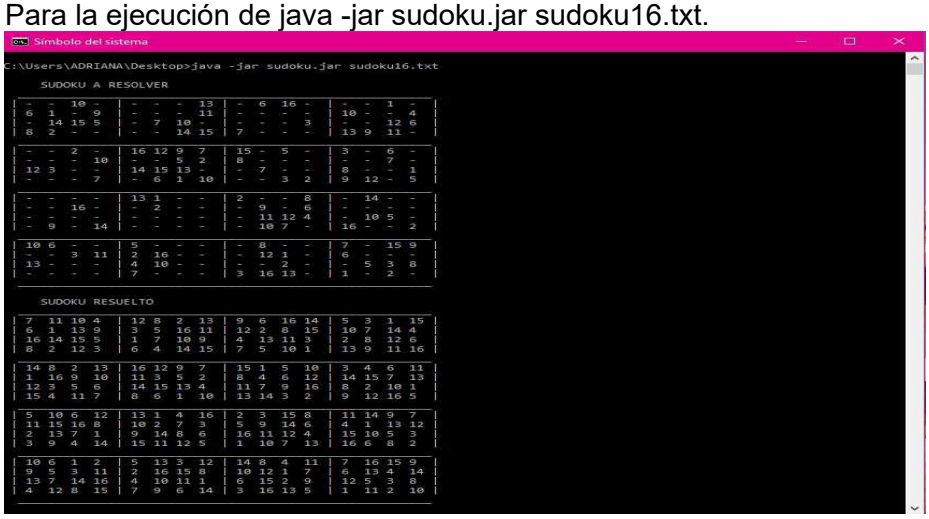
Un espacio de separación entre cada número o guión, dos espacios para marcar el cambio de celda y un intro (linea vacia) para marcar la celda

Para la ejecución de java -jar sudoku.jar sudoku4.txt.

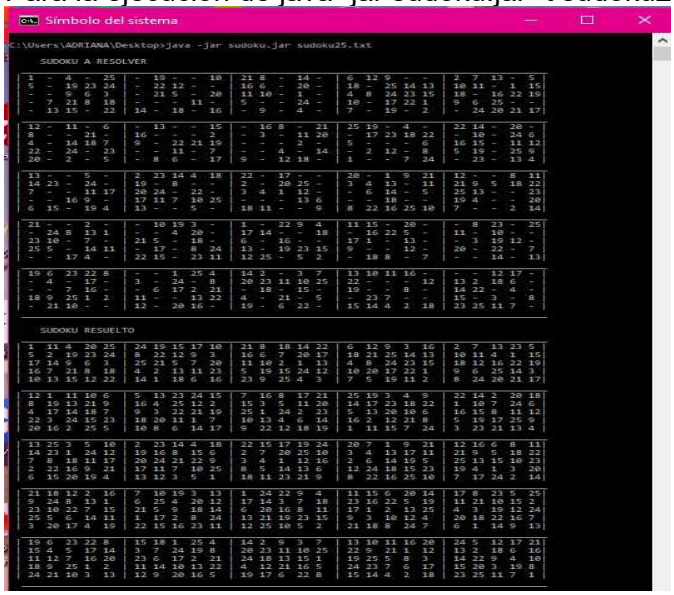


Para la ejecución de java -jar sudoku.jar sudoku9.txt.

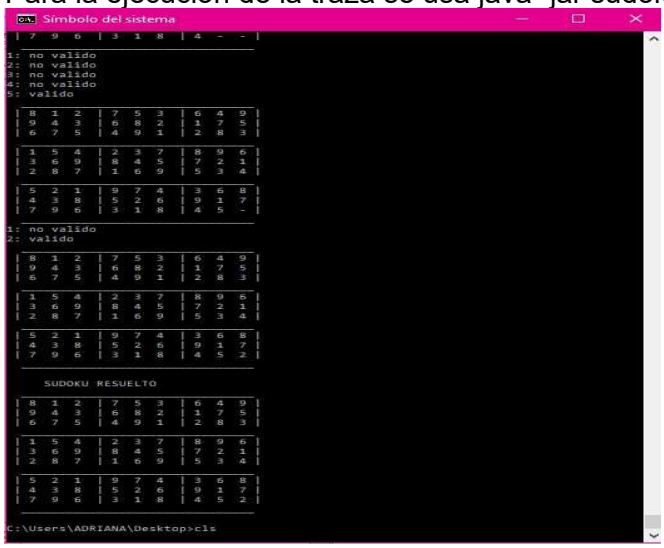
Para la ejecución de java -jar sudoku.jar sudoku16.txt.



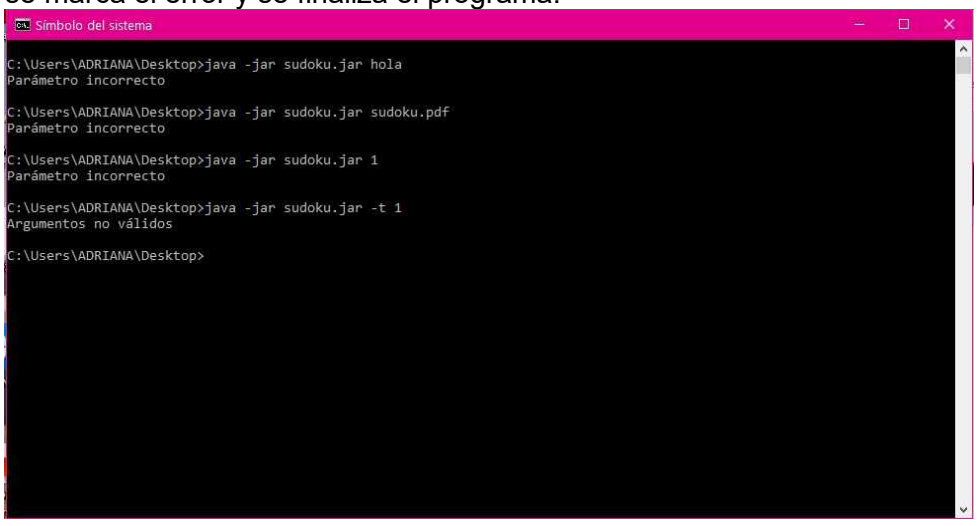
Para la ejecución de java -jar sudoku.jar -t sudoku25.txt.



Para la ejecución de la traza se usa java -jar sudoku.jar -t sudoku9.txt.



Y para otros casos, como por ejemplo, otros formatos de archivo, o otros argumentos se marca el error y se finaliza el programa.



Código fuente

Clase hazsudoku.java:

```
import java.io.BufferedReader;
import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;


public class hazsudoku {


    int lineas = 0; //lineas que se leen del archivo en caso de
haberlo


   //función para leer archivo

   public String[][] leeArchivo (File archivo) throws
FileNotFoundException, IOException {

      String cadena;

      int a=0,c=0;

        FileReader f = new FileReader(archivo);

        BufferedReader b = new BufferedReader(f);

        //primero contamos las lineas

        while((cadena = b.readLine())!=null) {

         if (!cadena.isEmpty()) {

               lineas = lineas+1;

         }

        }

        b.close();

        String matriz[][] = new String[lineas][lineas];

        try {

           //volvemos a leer el archivo

           FileReader f2 = new FileReader(archivo);

           BufferedReader b2 = new BufferedReader(f2);

           while((cadena = b2.readLine())!=null) {

            if (!cadena.isEmpty()) {

                  String[] values = cadena.split(" ");
```

```
                //recorremos el array de string
                for (int i=0; i<values.length; i++) {
                   if (!values[i].isEmpty()) {
                      matriz[a][c] = values[i];
                      c++;
                   }
                }
                c=0;
                a++;
          }
         }
    } catch (IOException | NumberFormatException e) {
        e.printStackTrace();
    }

   return matriz;
}

 /*función que coge la cuadricula medio completa e intenta
  * asignar valores en todos los huecos sin duplicar números
  * en filas, columnas y cuadros.*/
 public boolean resolverSudoku(int[][] matriz) {

   int fila = -1;
     int col = -1;
     boolean vacio = true;
     for (int i=0; i<matriz.length; i++) {
         for (int j=0; j<matriz.length; j++) {
             if (matriz[i][j] == 0) {
               fila = i;
                 col = j;
                 /*todavia quedan valores*/
                 vacio = false;
                 break;
```

```
            }
        }
        if (!vacio) {
            break;
        }
    }

    /*no hay ningun espacio vacio*/
    if (vacio) {
        return true;
    }

    /*vuelta atrás para cada fila*/
    for (int num=1; num<=matriz.length; num++) {
        if (esSeguro(matriz, fila, col, num)) {
           matriz[fila][col] = num;
            if (resolverSudoku(matriz)) {
                return true;
            }
            else {
                /*reemplazamos*/
              matriz[fila][col] = 0;
            }
        }
    }
    return false;
}

/*funcion para dibujar el sudoku*/
public void dibujarSudoku(int[][] matriz) {
  int i, j, grado = (int) Math.sqrt(lineas);
  String rayas = "";
  if (grado==2) {
     rayas="  _____________";
```

```
        }
        else if(grado==3) {
           rayas="  ____________________________";
        }
        else if(grado==4) {
           rayas="
_____________________________________________________";
        }
        else if(grado==5) {
           rayas="
_____________________________________________________________________
_____________";
        }
        else if(grado==6) {
           rayas="  __________";
        }
        else if(grado==7) {
           rayas="  __________";
        }
        else if(grado==8) {
           rayas="  __________";
        }
        else if(grado==9) {
           rayas="  __________";
        }
        else if(grado==10) {
           rayas="  __________";
        }

        System.out.println(rayas);
        for (i=0; i<matriz.length; i++) {
              for (j=0; j<matriz.length; j++) {
                 if (matriz[i][j]==0) {
                    if (j==0) {
                       System.out.print(" | ");
                    }
```

```
System.out.print("-");
if (j==grado-1) {
   System.out.print("  | ");
}
else if (grado>2 && j==(grado*2)-1) {
System.out.print("  | ");
}
else if (grado>3 && j==(grado*3)-1) {
System.out.print("  | ");
}
else if (grado>4 && j==(grado*4)-1) {
System.out.print("  | ");
}
else if (grado>5 && j==(grado*5)-1) {
System.out.print("  | ");
}
else if (grado>6 && j==(grado*6)-1) {
System.out.print("  | ");
}
else if (grado>7 && j==(grado*7)-1) {
System.out.print("  | ");
}
else if (grado>8 && j==(grado*8)-1) {
System.out.print("  | ");
}
else if (grado>9 && j==(grado*9)-1) {
System.out.print("  | ");
}
else if (grado>10 && j==(grado*10)-1) {
System.out.print("  | ");
}
else if (j==matriz.length-1) {
System.out.print(" | ");
}
else {
```

```
            System.out.print("  ");
        }
    }
    else if (matriz[i][j]>9) {
        if (j==0) {
            System.out.print(" | ");
        }
        System.out.print(matriz[i][j]);
        if (j==grado-1) {
            System.out.print(" | ");
        }
        else if (grado>2 && j==(grado*2)-1) {
            System.out.print(" | ");
        }
        else if (grado>3 && j==(grado*3)-1) {
            System.out.print(" | ");
        }
        else if (grado>4 && j==(grado*4)-1) {
            System.out.print(" | ");
        }
        else if (grado>5 && j==(grado*5)-1) {
            System.out.print(" | ");
        }
        else if (grado>6 && j==(grado*6)-1) {
            System.out.print(" | ");
        }
        else if (grado>7 && j==(grado*7)-1) {
            System.out.print(" | ");
        }
        else if (grado>8 && j==(grado*8)-1) {
            System.out.print(" | ");
        }
        else if (grado>9 && j==(grado*9)-1) {
            System.out.print(" | ");
        }
```

```
            else if (grado>10 && j==(grado*10)-1) {
                System.out.print(" | ");
            }
            else if (j==matriz.length-1) {
            System.out.print("| ");
            }
            else {
                System.out.print(" ");
            }
        }
        else {
            if (j==0) {
                System.out.print(" | ");
            }
            System.out.print(matriz[i][j]);
            if (j==grado-1) {
                System.out.print("  | ");
            }
            else if (grado>2 && j==(grado*2)-1) {
            System.out.print("  | ");
            }
            else if (grado>3 && j==(grado*3)-1) {
            System.out.print("  | ");
            }
            else if (grado>4 && j==(grado*4)-1) {
            System.out.print("  | ");
            }
            else if (grado>5 && j==(grado*5)-1) {
            System.out.print("  | ");
            }
            else if (grado>6 && j==(grado*6)-1) {
            System.out.print("  | ");
            }
            else if (grado>7 && j==(grado*7)-1) {
            System.out.print("  | ");
```

```
            }
            else if (grado>8 && j==(grado*8)-1) {
            System.out.print("  | ");
            }
            else if (grado>9 && j==(grado*9)-1) {
            System.out.print("  | ");
            }
            else if (grado>10 && j==(grado*10)-1) {
            System.out.print("  | ");
            }
            else if (j==matriz.length-1) {
            System.out.print(" | ");
            }
            else {
               System.out.print("  ");
            }
         }
      }
      if (i==grado-1) {
         System.out.println("\n"+rayas);
      }
   else if (grado>2 && i==(grado*2)-1) {
         System.out.println("\n"+rayas);
   }
   else if (grado>3 && i==(grado*3)-1) {
         System.out.println("\n"+rayas);
   }
   else if (grado>4 && i==(grado*4)-1) {
         System.out.println("\n"+rayas);
   }
   else if (grado>5 && i==(grado*5)-1) {
         System.out.println("\n"+rayas);
   }
   else if (grado>6 && i==(grado*6)-1) {
         System.out.println("\n"+rayas);
```

```
            }
            else if (grado>7 && i==(grado*7)-1) {
                  System.out.println("\n"+rayas);
            }
            else if (grado>8 && i==(grado*8)-1) {
                  System.out.println("\n"+rayas);
            }
            else if (grado>9 && i==(grado*9)-1) {
                  System.out.println("\n"+rayas);
            }
            else if (grado>10 && i==(grado*10)-1) {
                  System.out.println("\n"+rayas);
            }
            else if (i==matriz.length-1) {
                  System.out.println("\n"+rayas);
            }
               else {
                  System.out.println();
               }
            }
      }

      /*comprobamos si es seguro asignar un número a la fila, columna dada*/
      public boolean esSeguro(int[][] matriz, int fila, int col, int num) {
        int grado = (int) Math.sqrt(lineas);

          /*comprobamos si encontramos el mismo número en la fila similar y devolvemos false */
          for (int x=0; x<=matriz.length-1; x++)
              if (matriz[fila][x] == num)
                  return false;

          /*comprobamos si encontramos el mismo número en la columna similar y devolvemos false */
```

```
        for (int x=0; x<=matriz.length-1; x++)
            if (matriz[x][col] == num)
                return false;


        /*comprobamos si encontramos el mismo número en la matriz 3*3
y devolvemos false */
        int filaInicio = fila-fila%grado, columnaInicio = col-col
%grado;
        for (int i=0; i<grado; i++)
            for (int j=0; j<grado; j++)
                if (matriz[i+filaInicio][j+columnaInicio] == num)
                    return false;
        return true;
    }


    public boolean dibujarTraza(int[][] matriz) {
      dibujarSudoku(matriz);
      int fila = -1;
        int col = -1;
        boolean vacio = true;
        for (int i=0; i<matriz.length; i++) {
            for (int j=0; j<matriz.length; j++) {
                if (matriz[i][j] == 0) {
                  fila = i;
                    col = j;
                    /*todavia quedan valores*/
                    vacio = false;
                    break;
                }
            }
            if (!vacio) {
                break;
            }
        }
```

```
        /*no hay ningun espacio vacio*/
        if (vacio) {
            return true;
        }

        /*vuelta atrás para cada fila*/
        for (int num=1; num<=matriz.length; num++) {
         System.out.print(num+": ");
            if (esSeguroTraza(matriz, fila, col, num)) {
               matriz[fila][col] = num;
                if (dibujarTraza(matriz)) {
                    return true;
                }
                else {
                    /*reemplazamos*/
                  matriz[fila][col] = 0;
                }
            }
            else {
               System.out.println("no valido");
            }
        }
        return false;
    }

    /*comprobamos si es seguro asignar un número a la fila, columna
dada*/
    public boolean esSeguroTraza(int[][] matriz, int fila, int col,
int num) {
      int grado = (int) Math.sqrt(lineas);

        /*comprobamos si encontramos el mismo número en la fila
similar y devolvemos false */
        for (int x=0; x<=matriz.length-1; x++)
            if (matriz[fila][x] == num)
```

```
                return false;


        /*comprobamos si encontramos el mismo número en la columna
similar y devolvemos false */
        for (int x=0; x<=matriz.length-1; x++)
            if (matriz[x][col] == num)
                return false;


        /*comprobamos si encontramos el mismo número en la matriz 3*3
y devolvemos false */
        int filaInicio = fila-fila%grado, columnaInicio = col-col
%grado;
        for (int i=0; i<grado; i++)
            for (int j=0; j<grado; j++)
                if (matriz[i+filaInicio][j+columnaInicio] == num)
                    return false;
      System.out.println("valido");
        return true;
    }
}
```

Clase sudoku.java:

```
import java.io.File;
import java.io.IOException;

import java.nio.file.Files;


public class sudoku {

   //función para saber si un valor es numérico

   public static boolean isNumeric(String cadena) {

        boolean resultado;

        try {

            Integer.parseInt(cadena);

            resultado = true;

        } catch (NumberFormatException excepcion) {

            resultado = false;

        }

        return resultado;

    }


   public static boolean isDoubleInt(double d)  {

          //select a "tolerance range" for being an integer

          double TOLERANCE = 1E-5;

          //do not use (int)d, due to weird floating point
conversions!

          return Math.abs(Math.floor(d) - d) < TOLERANCE;

   }


    public static void main(String[] args) {

      hazsudoku sudoku = new hazsudoku(); //Objeto de la clase sudoku
para hacer las funciones


      //si no hay parametros

      if(args.length == 0){

         String m[][] = { { "5", "3", "-", "-", "7", "-", "-", "-",
"-" },

                  { "6", "-", "-", "1", "9", "5", "-", "-", "-" },

                  { "-", "9", "8", "-", "-", "-", "-", "6", "-" },
```

```
                { "8", "-", "-", "-", "6", "-", "-", "-", "3" },
                { "4", "-", "-", "8", "-", "3", "-", "-", "1" },
                { "7", "-", "-", "-", "2", "-", "-", "-", "6" },
                { "-", "6", "-", "-", "-", "-", "2", "8", "-" },
                { "-", "-", "-", "4", "1", "9", "-", "-", "5" },
                { "-", "-", "-", "-", "8", "6", "-", "7", "9" }
            };


    //convertimos la matriz de caracteres a matriz de enteros
    int matriz[][] = new int[m.length][m.length];

    for (int f=0; f<m.length; f++){
       for (int c=0; c<m.length; c++) {
          if(!isNumeric(m[f][c]))
             matriz[f][c] = 0;
          else
             matriz[f][c] = Integer.parseInt(m[f][c]);
       }
    }

      System.out.println("\n     SUDOKU A RESOLVER");
      sudoku.dibujarSudoku(matriz);

    if (sudoku.resolverSudoku(matriz)) {
          System.out.println("\n     SUDOKU RESUELTO");
          sudoku.dibujarSudoku(matriz);
    }
    else {
        System.out.println("No existe solución");
    }
}

//si hay un parámetro
```

```
      else if(args.length == 1){
         String valor = args[0];
         File archivo = new File(valor);
         if (valor.equals("-t")) {
            String m[][] = { { "5", "3", "-", "-", "7", "-", "-", "-",
"-" },
                     { "6", "-", "-", "1", "9", "5", "-", "-",
"-" },
                     { "-", "9", "8", "-", "-", "-", "-", "6",
"-" },
                     { "8", "-", "-", "-", "6", "-", "-", "-",
"3" },
                     { "4", "-", "-", "8", "-", "3", "-", "-",
"1" },
                     { "7", "-", "-", "-", "2", "-", "-", "-",
"6" },
                     { "-", "6", "-", "-", "-", "-", "2", "8",
"-" },
                     { "-", "-", "-", "4", "1", "9", "-", "-",
"5" },
                     { "-", "-", "-", "-", "8", "6", "-", "7", "9" }
                   };


            //convertimos la matriz de caracteres a matriz de enteros
            int matriz[][] = new int[m.length][m.length];

            for (int f=0; f<m.length; f++){
               for (int c=0; c<m.length; c++) {
                  if(!isNumeric(m[f][c]))
                     matriz[f][c] = 0;
                  else
                     matriz[f][c] = Integer.parseInt(m[f][c]);
               }
            }

              System.out.println("\n     SUDOKU A RESOLVER");
```

```
                sudoku.dibujarSudoku(matriz);


            if (sudoku.dibujarTraza(matriz)) {
                  System.out.println("\n     SUDOKU RESUELTO");
                  sudoku.dibujarSudoku(matriz);
            }
            else {
                System.out.println("No existe solución");
            }
         }
         else if(valor.equals("-h")) {
            System.out.println("SINTAXIS: sudoku [-t][-h] [fichero
entrada]\n"
                  + "-t Traza cada llamada recursiva y sus
parámetros.\n"
                  + "-h Muestra esta ayuda\n"
                  + "[fichero entrada] Tabla inicial del Sudoku");
              System.exit(0);
         }
         //comprobar si existe fichero en la ruta actual
         else if (archivo.exists()) {
             try {
                String tipodeArchivo =
Files.probeContentType(archivo.toPath());
              if(tipodeArchivo.equals("text/plain")) {
                 String m[][] = sudoku.leeArchivo(archivo);
                 //tiene que haber más de un jugador
                    if(!isDoubleInt(Math.sqrt(sudoku.lineas)) ||
Math.sqrt(sudoku.lineas)>10 || Math.sqrt(sudoku.lineas)<2) {
                    System.out.println("Solo se admiten sudokus de
los cuadrados de 2,3,4,5,6,7,8,9 y 10\n\n");
                       System.exit(0);
                    }
                    else {
                    //convertimos la matriz de caracteres a matriz de
enteros
```

```
                    int matriz[][] = new int[sudoku.lineas]
[sudoku.lineas];

                    for (int f=0; f<sudoku.lineas; f++){
                       for (int c=0; c<sudoku.lineas; c++) {
                          if(!isNumeric(m[f][c]))
                             matriz[f][c] = 0;
                          else
                             matriz[f][c] = Integer.parseInt(m[f]
[c]);
                       }
                    }

                      System.out.println("\n     SUDOKU A RESOLVER");
                      sudoku.dibujarSudoku(matriz);

                    if (sudoku.resolverSudoku(matriz)) {
                          System.out.println("\n     SUDOKU
RESUELTO");
                          sudoku.dibujarSudoku(matriz);
                    }
                    else {
                        System.out.println("No existe solución");
                    }
                 }
              }
           }
           catch (IOException ioException) {
               System.out.println("Error: " +
ioException.getMessage());
                System.exit(0);
           }
        }
        //parametro incorrecto
        else {
              System.out.println("Parámetro incorrecto");
```

```
                System.exit(0);
            }
        }

      //si hay dos parámetros
      else if(args.length == 2){
         String valor1 = args[0];
         String valor2 = args[1];
         File archivo = new File(valor2);
         if (valor1.equals("-t") && archivo.exists()) {
            try {
                String tipodeArchivo =
Files.probeContentType(archivo.toPath());
               if(tipodeArchivo.equals("text/plain")) {
                  String m[][] = sudoku.leeArchivo(archivo);
                  //tiene que haber más de un jugador
                     if(!isDoubleInt(Math.sqrt(sudoku.lineas)) ||
Math.sqrt(sudoku.lineas)>10 || Math.sqrt(sudoku.lineas)<2) {
                     System.out.println("Solo se admiten sudokus de
los cuadrados de 2,3,4,5,6,7,8,9 y 10\n\n");
                        System.exit(0);
                     }
                  else {

                     //convertimos la matriz de caracteres a matriz de
enteros
                     int matriz[][] = new int[sudoku.lineas]
[sudoku.lineas];

                     for (int f=0; f<sudoku.lineas; f++){
                        for (int c=0; c<sudoku.lineas; c++) {
                           if(!isNumeric(m[f][c]))
                              matriz[f][c] = 0;
                           else
                              matriz[f][c] = Integer.parseInt(m[f]
[c]);
                        }
```

```
                    }

                      System.out.println("\n     SUDOKU A RESOLVER");
                      sudoku.dibujarSudoku(matriz);

                    if (sudoku.dibujarTraza(matriz)) {
                          System.out.println("\n     SUDOKU
RESUELTO");
                          sudoku.dibujarSudoku(matriz);
                    }
                    else {
                        System.out.println("No existe solución");
                    }
                  }
                }
              catch (IOException ioException) {
                  System.out.println("Error: " +
ioException.getMessage());
                   System.exit(0);
              }
          }
          else {
                System.out.println("Argumentos no válidos");
                System.exit(0);
          }
      }

      //si hay más de dos parámetro
      else {
            System.out.println("Solo se permiten 2 parámetros de
entrada como máximo");
            System.exit(0);
        }
    }
```

```
}
```