# MEMORIA PRÁCTICA DE PROGRAMACIÓN III (Curso 2009-2010)

**Diego J. Martínez García**
**666659277**
**apussapus@gmail.com**

Indice



## 1- Descripción y justificación del tipo de Algoritmo Utilizado

Puesto que el tomar la decisión de tender un puente afecta tanto a las celdas vecinas como a otras celdas distantes, vemos que el problema no puede ser descompuesto en otros de menor tamaño, por lo que no puede usarse el esquema de Divide y Vencerás.

La resolución a mano de algún hashiwokakero sencillo pronto nos hace comprender que se trata de un juego de ensayo y error en el que no vasta aplicar una serie de reglas ciegas como en el caso del esquema Voraz ya hay que estar muy atento al contexto del problema.

Esto, unido a que nos encontramos ante un grafo en el que solo hace falta encontrar una única solución y no hay condiciones de optimización, nos induce a pensar que el enfoque correcto para este problema es el del algoritmo de Vuelta Atrás.

2- Glosario de términos

- **Hashi**: Los objetos de esta clase representan la situación del tablero en un momento dado mediante un par de conjuntos de puentes (verticales y horizontales) y un par de listas, ListaDeIslas y listaDeCandidatas.
- **Isla**: Objeto con los atributos fila, columna y valor que representa a uno de los valores numéricos de la cuadrícula.
- **ListaDeIslas**: Es un atributo de la clase Hashi que tiene forma de ArrayList<Isla> donde se guardan las islas del Hashi cuyo valor sea distinto de cero. El algoritmo de resolución disminuye en uno el valor de las islas que acaban de formar un puente y va quitando de la ListaDeIslas a las islas cuyo valor llega a cero, así cuando esla ListaDeIslas llega a cero se ha llegado a una solución del Hashi.
- Las islas "**Agotadas**" son las que han tendido todos sus puentes y ya no aparecen en la ListaDeIslas del Hashi.
- Las **islas vecinas** a la isla considerada son con las que se está en condiciones de tender algun puente a partir de la isla dada.
- **Isla candidata**: Una isla que ya tiene tendido algún puente y es susceptible de tender alguno más. El algoritmo de resolución utilizado va tendiendo puentes que salen de la lista de islas candidatas, favoreciendo así la formación de la única región conexa que nos exige el enunciado del problema.
- **Isla afortunada**: Entiendo por "Isla Afortuanada" a aquella a la que le es indiferente la direccion que tome su proximo puente, tiende todos sus puentes hacia las vecinas de manera forzada. Esta rodeada por una serie de vecinas y a todas tiene que tender al menos un puente.
- Entiendo por "**Isla semi Afortuanada**" a aquella a la que le es indiferente la direcciones que tomen sus puentes, menos el último , que queda indefinido.
  Se da en el caso de islas con valor impar
  P.ej. una isla de valor 3 rodeada por dos vecinas, una de valor 5 rodeada por 3 vecinas y una de valor 7 rodeada por 4 vecinas

Ejemplo

| Hashi Inicial | Una posible resultante | Lista de Islas del Hashi Resultante | Lista de Candidatas |
|---|---|---|---|
| 3 4 2<br><br>1 1 2 | 1 = * – 1<br>\|<br>\|<br>1 * 2 | Coordenadas<br>(0,0),(0,3),(1,0),(1,3)<br><br>Valores<br>1, 1, 1, 2 | 1 = * – 1<br>\|<br>\|<br>1 * 2 |

- **Puente**: Objeto que representa el puente tendido entre dos islas, sus atributos son las islas implicadas, el tipo de puente (simple o doble), su orientación (horizontal o vertical) y un par de arrays donde guardamos las filas y las columnas por las que atraviesa el puente.

3- Estrategias locales consideradas y condiciones de poda

- inicialmenteValido() : Para que un hashi sea inicialmente valido el valor acumulado de sus islas en la lista ha de ser par.

- parcialmenteValido(): Un Hashi sera parcialmente valido cuando:
  El valor de una isla (el num de puentes a tender a partir de una isla) sea menor o igual al num de puentes posibles con sus vecinas a partir de esa isla en el momento actual.
  Con esto evito situaciones del tipo : (2)--(2) y (1)--(3)--(1) donde el número de puentes que se pueden tender es menor al necesario para dejar a cero los valores de las islas.

- PotencialmenteConexo(): Un hashi es potencialmenteConexo cuando las islas que ya no están en la lista de islas (las islas "Agotadas" que han tendido todos sus puentes) PUEDEN SER conectadas mediante algún camino a alguna isla que aún está en la lista de Islas. De no ser así se estarían formando regiones inconexas, incompatibles con la solución correcta del problema. Utilizo un recorrido en anchura basado en la estructura de datos de cola para implementar este método.

4- Heurísticas

- La clase Tratamiento: Esta clase se encarga de tratar los Hashis realizando sucesivas pasadas en busca de islas afortunadas y
  semi afortunadas para tender los puentes de las mismas y devolver al hashi ya tratado.
  Esta clase se utiliza en dos contextos:

  1- Con el método "generarHashiInicial" ,al iniciar el programa para realizar el pretratamiento del tablero inicial en busca de afortunadas antes de pasárselo al algoritmo de vuelta atras ,
  esto puede facilitarle mucho las cosas al vuelta atrás pues puede que este tratamiento simplifique sobremanera el tablero inicial.

  2- Dentro del algoritmo vuelta atrás cuando se genera un ensayo tendiendo un puente entre las islas se llama seguidamente al
  método "tratarHashi" para que compruebe si el nuevo puente ha provocado la creación de nuevas afortunadas o semiafortunadas.

- En el método islasVecinas ( isla):

  1- si la vecina mirada no tiene a su vez vecinas, el puente con la isla original es obligado y por tanto , para favorecer cuanto antes su formación, lo pongo al principio de la lista de vecinas
  2 - para favorecer la creación de puentes simples frente a los dobles
  Si no hay ningún puente previo entre ambas, la isla mirada se pone al principio de la lista de vecinas.

- En el método tenderPuente(islaA, islaB):
  Para evitar la rapida formacion de puentes dobles:
  si alguna de estas islas solo puede tender puentes dobles (tiene ya puentes con todas

sus vecinas ) la pongo al final de la lista de candidatas, recolocando la lista de candidatas.

- En el método recolocaCandidatas():

Para evitar la rapida formacion de puentes dobles:
voy revisando la lista de las candidatas y si alguna de estas islas solo puede tender puentes dobles (tiene ya puentes con todas sus vecinas ) la pongo al final de la lista de candidatas.

- En el método compleciones( hashi):

Ordeno la lista de hashis nuevos según la longitud de su lista de canditatas de menor a mayor, generalmente la accion de poner primero las compleciones mas restrictivas hace más eficiente la vuelta atras (wikipedia).

-En el método vueltaAtras(hashi, myEntSal, depuracion):

Hago una revisión de los avances del algoritmo cada diez nodos: si no se ha disminuido significativamente el numero de islas (si no ha disminuido la cota) es que no hay avances y entonces recomienzo con el siguiente hashi de las compleciones del hashi inicial. Si por el contrario si se han producido avances, actualizo el valor de la cota , disminuyéndolo al numero de islas actual

## 5- Análisis del coste

En el estudio del coste en los algoritmos de vuelta atrás es difícil dar una solución exacta, ya que se desconoce el tamaño de lo explorado a priori, (puede que las heurísticas funcionen muy bien en la mayoria de los casos, pero que no lo hagan en absoluto en algún otro) , así que solo es posible en la mayoría de los casos dar una cota superior al tamaño de la búsqueda.

Así que ,tal y como se ha construido el programa , se considera que en cada nodo del árbol de búsqueda se tiene un tablero hashi , y para el tablero selecciono una isla candidata, y a partir de esa isla tiendo un puente con alguna de sus vecinas .

Dado que una isla tiene un máximo de cuatro vecinas tengo cuatro posibilidades de tender un puente para una candidata , suponiendo que todas las n islas del hashi fuesen candidatas tendríamos $4^n$ posibilidades, lo que nos da un coste exponencial para el algoritmo, (en el caso peor).
Este factor exponencial querría decir que el problema es irresoluble, en la práctica , para valores grandes de n.

En general se puede considerar que si $\beta$ es el índice de ramificación del árbol y $\gamma$ es el número de elementos que componen una solución, o lo que es lo mismo, la profundidad del arbol de búsqueda, vamos en realidad a tener como máximo $\beta^{\gamma}$ pasos para alcanzar todas las soluciones. Si además se introducen condiciones de poda, éstas influyen reduciendo el índice de ramificación. La poda además dependerá de la profundidad p y del número n de piezas disponibles para completar la solución. El coeficiente de poda es un valor entre 0 y 1, si el coeficiente de poda es 0, no existen condiciones de poda y la búsqueda es totalmente ciega, si es 1 no habrá ramificación. La expresión general sería $[\ \beta\ (1 - \zeta(n , p))]^{\gamma}$. En este problema $\beta = 4$ , $\gamma = n$ y $\zeta$ es una función que varía entre 0 y $\beta - 1 / \beta$.

## 6- Casos de Prueba

Caso de ejecución negativo (el número de islas del hashi no es par)

```
* 3 * 4 * 2 *
* * * * 3 * 3
* * * * * * *
* * * * * * *
* 4 * * 3 * 3
* * * * * * *
1 * 3 * * * 4
```

*ejemploHashi7_7negativo.txt*

```
Pretratando el tablero inicial ...
/

No existe solucion ya que el hashi esta mal construido.
```

Salida del programa

Caso de ejecución negativo (que tiene una solución pero de tipo inconexo)

```
1 * 4 * 3 * * 2
* * * * * * * *
2 * 3 * * 2 * *
* * * * * * * 1
3 * * * * 3 * *
* * * * * * * *
* * * * * * * *
* * * * 2 * * 2
```

*ejemploHashi8_8_negativo1.txt*

```
1   4 = 3 - - 2
|   H         |
2   3 - - 2   |
|           |   1
3 = = = = 3

          2 = = 2
```

*Posible solución errónea inconexa*

```
Pretratando el tablero inicial ...
///
Procesando nodos con el algoritmo de vuelta atras ...


El hashi no tiene solucion conexa, se ha explorado todo el arbol del juego
 sin encontrala.
```

*Salida del programa,*

Caso de ejecución negativo (el hashi es imposible de resolver)

```
2 * 4 * 2 * * 2
* * * * * * * *
2 * 2 * * 2 * *
* * * * * * * 2
3 * * * * 3 * *
* * * * * * * *
* * * * * * * *
* * 3 * 3 * * 2
```

*ejemploHashi8_8_negativo2.txt*

Aunque el número de islas es par , la situación de las dos islas marcadas en naranja hace imposible su resolución ya que, para que la hubiese, sus puentes deberían cruzarse.

```
Pretratando el tablero inicial ...
/

No existe solucion ya que el hashi esta mal construido.
```

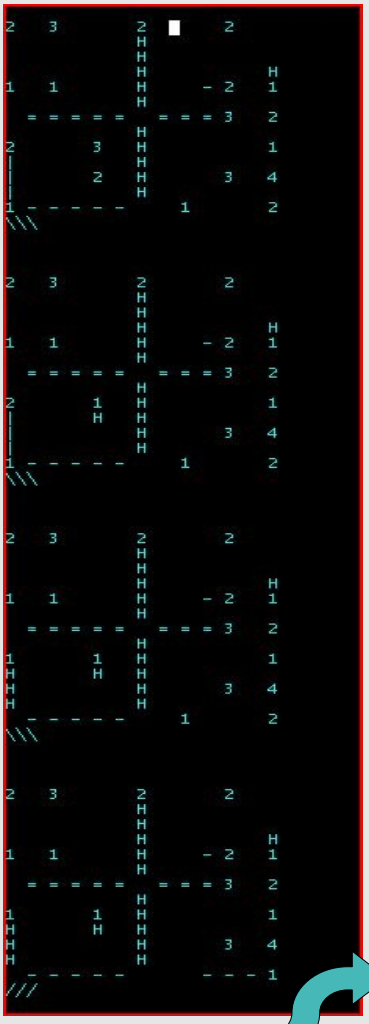
*Salida del programa*

En el programa se ha añadido una clase llamada **BancoDePruebas** en la que hay 40 hashis de todo tipo que pueden ser testeados en serie por el programa, sin más que teclear “java hashiwokakero -b” en el terminal.
La realización de esta batería de pruebas en mi computadora tarda sobre unos 18 minutos, dando una velocidad media de ~30seg por hashi.

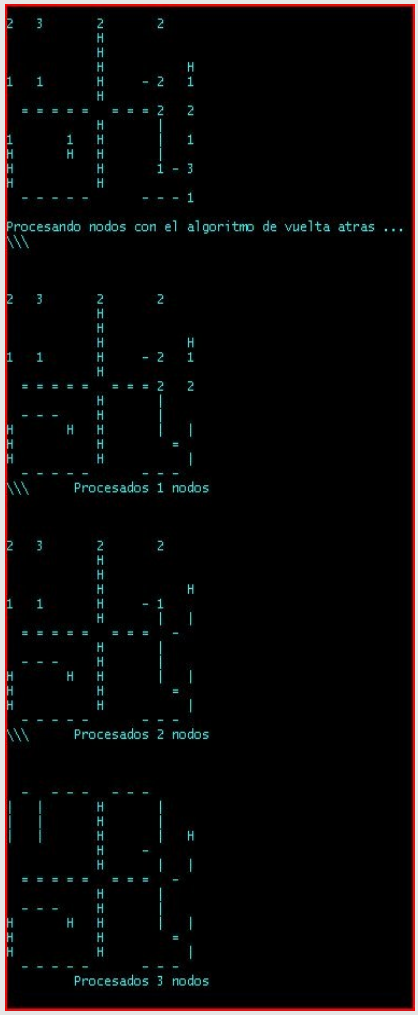
Caso de ejecución del ejemplo oficial de la práctica en modo traza

*(1) Tablero inicial y pretratamiento de los dos primeros nodos*

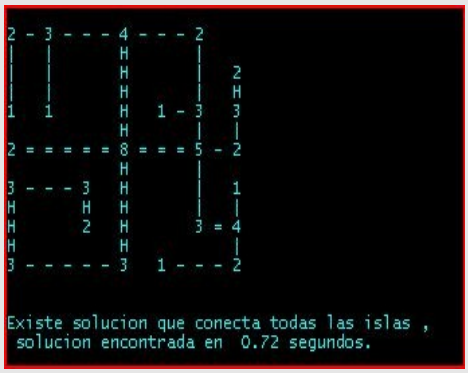

*(2) Continua el pretratamiento*



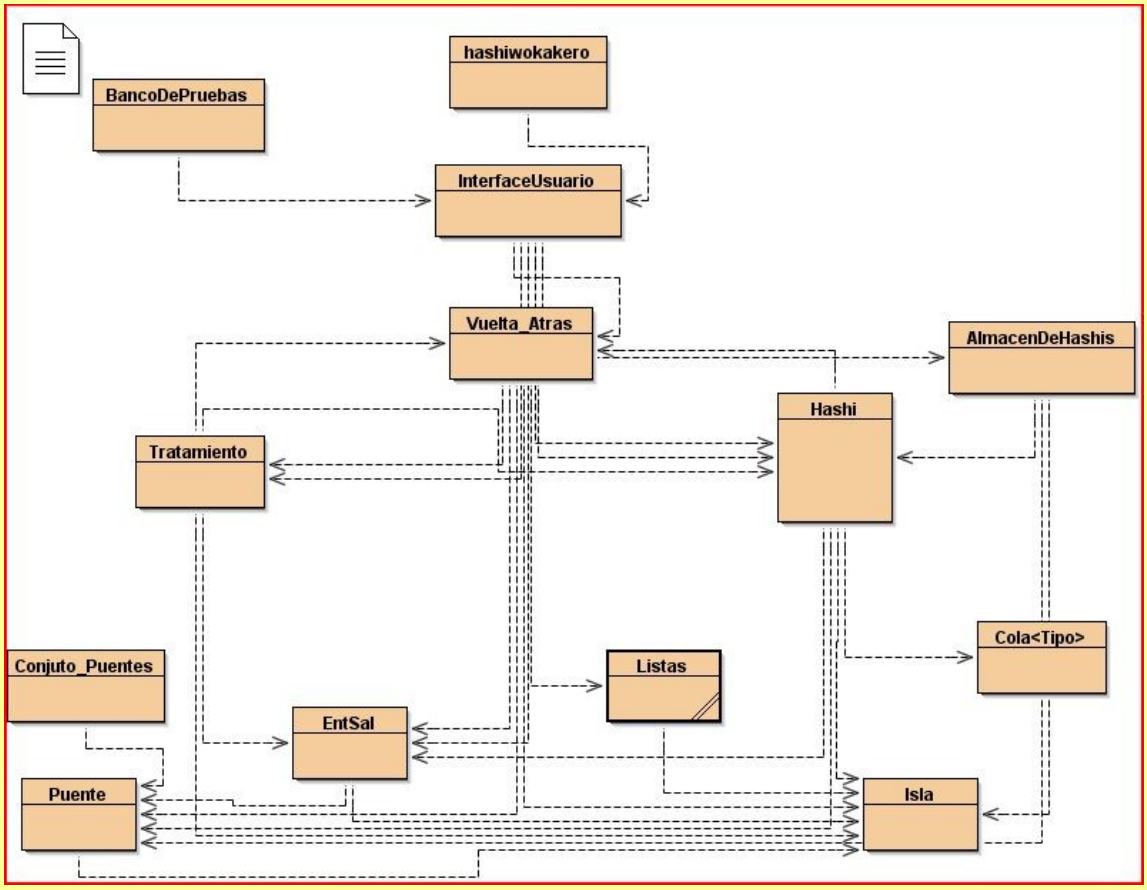
*(3) Final del pretratamiento e inicio del algoritmo de vuelta atrás*




*(4) Solución final*



7- Listado completo del código fuente




*Esquema jerárquico de las clases, las clases centrales Tratamiento, Hashi y Vuelta_Atras son las clases vitales para la resolución.*

## - Archivo Readme -

---

README file.

---

PROJECT TITLE: Hashiwokakero

PURPOSE OF PROJECT: Resolver todo tipo de puzzles Hashiwokakero de un tamaño
de 7 x 7 hasta 25x25

VERSION or DATE: 21 de Julio del 2010

HOW TO START THIS PROJECT: En el directorio donde se encuentre el programa
teclee en el terminal "java hashiwokakero ficheroDelPuzzle.txt"
donde ficheroDelPuzzle es el archivo de texto donde
guarda el Hashiwokakero a resolver

AUTHORS: Diego J. Martinez Garcia
Centro asociado de Almería
666659277
apussapus@gmail.com

USER INSTRUCTIONS:

Este programa puede ejecutarse con las siguientes sintaxis:

hashiwokakero [-t][-h] [fichero]

o bien

hashiwokakero -l numero fichero

o bien

hashiwokakero -b

ejemplos: 'hashiwokakero -t  fichero.txt'
'hashiwokakero -l 1000 fichero.txt'
'hashiwokakero -h'
'hashiwokakero -b'

Opciones:

t: Modo traza. Muestra toda la secuencia de resolucion del hashi
mostrando los tableros intermedios .

b: Modo bateria de pruebas  realiza la resolucion de 41 hashis
uno tras otro  de manera continua e imprime un resumen de los tiempos .

l: Modo limite. si se activa este modo el siguiente argumento ha de
ser un numero que marcara el limite de nodos de la vuelta atras
el valor por defecto de l es de 2500 nodos

h: Modo ayuda. Muestra la sintaxis y los créditos.

Si no tiene argumentos de seleccion de modo  el programa muestra la
matriz final resuelta  o un mensaje si no la encuentra.

Ello permite tambien usar redirecciones y 'pipes' como entrada de
datos.

ejemplos:
'hashiwokakero < ejemploHashiOficial.txt'
'type ejemploHashiOficial.txt | hashiwokakero'

ejemplo de un hashiwokakero:

```
#------------------------
# Ejemplo hashiwokakero
# Fuente: Nikoli
# Valores de la Figura 1
# ------------------------
```

```
##
13
13
2 * 3 * * * 4 * * * 2 * *
* * * * * * * * * * * * *
* * * * * * * * * * * * 2
* * * * * * * * * * * * *
1 * 1 * * * * * 1 * 3 * 3
* * * * * * * * * * * * *
2 * * * * * 8 * * * 5 * 2
* * * * * * * * * * * * *
3 * * * 3 * * * * * * * 1
* * * * * * * * * * * * *
* * * * 2 * * * * * 3 * 4
* * * * * * * * * * * * *
3 * * * * * 3 * 1 * * * 2
+
```

## - Clase Vuelta_Atras -

```
import  java.io.*;
import  java.util.*;

import java.lang.Cloneable;
import java.lang.Object;

/**
 * Esta es la clase  en la que implementamos el algoritmo de vuelta atras, utilizando los objetos
 *  de las demas clases
 */
public class Vuelta_Atras
{

   private int i;
   private int cota;

   private int longitudLista;

   private static final Comparator<Listas> comparador = new ListasOrdenadasPorLongitud();

   private  int nodos;
```

```
  // nodosLimite son el numero maximo de nodos que el algoritmo puede tratar antes de darse por vencido (vienen marcados por
  //  el usuario)
  private int nodosLimite;

  private  boolean acabado;

  private static class ListasOrdenadasPorLongitud implements Comparator<Listas>
  {
      public int compare(Listas t1, Listas t2)
      {
            return t1.compareTo(t2);
      }
   }

   /**
    * Constructor
    */
   public Vuelta_Atras()
   {

      nodos = 0;
      acabado = false;
      nodosLimite = 2500;
      i = 1;
   }
   /**
    * Constructor para el caso de trabajar con hashis
    */
   public Vuelta_Atras(int limite)
   {

      nodos = 0;
      acabado = false;
      nodosLimite = limite;
      i = 1;

   }

   public  int obtenerNodos(){

      return nodos;
   }
   public int obtenerNodosLimite(){

      return nodosLimite;
   }

   /**
    * Dado un Hashi de partida y un par de islas devuelve este hashi con las dos islas conectadas, además,
    * si el nuevo puente ha provocado la creación de islas afortunadas a su alrededor se tienden los
```

```
puentes de
     * estas afortunadas
     *
     * @param   hashiDePartida,  islaDePartida,  islaDeLlegada   el hashi y las dos islas iniciales
     * @return     hashiDePartida que ha sido modificado con los nuevos puentes, el metodo devuelve
null si el
     *             hashi a devolver no cumple con las reglas del juego
     */
    public  Hashi generar_ensayo(Hashi hashiDePartida, Isla islaDePartida, Isla islaDeLlegada)
    {

        //Tiendo el puente a partir del nuevo Hashi, las islas de partida y de llegada tienen que
        // pertenecer a la lista del Hashi de partida
        hashiDePartida.tenderPuente(islaDePartida, islaDeLlegada);

       //HEURISTICA: Tras tender el puente,  realizo un tratamiento al hashi resultante en busca de
posibles nuevas
       // islas afortunadas o semiafortunadas
       if(!Tratamiento.inicialmenteValido(hashiDePartida.obtenerListaDeIslas()))
           return null;

       EntSal myEntSal = new EntSal();
       Hashi hashiOriginal = hashiDePartida;
       Hashi hashiInicio = hashiDePartida;
       boolean nuevasAfortunadas = false;

       int pasada = 0;
       boolean depuracion = false;

        hashiInicio = Tratamiento.tratarHashi(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada,
myEntSal, depuracion);

       return hashiInicio;

    }

    /**
     *  Las compleciones del algoritmo de vuelta atras: Las compleciones de un hashi dado seran una
lista de hashis que
     *  se han generado a partir de el. Cada hashi nuevo generado se mira en la lista global de los
hashis que ya han
     *  sido generados "almacenListas", si no esta en ella se le mete y si ya estaba se le descarta para
las compleciones
     *  esto nos evita el tener que estar considerando hashis duplicados en las diferentes compleciones
     *
     *  @param   hashiDePartida   el hashi sobre el que se van a hacer las compleciones
     *  @return     listaHashisNuevos lista con todos los hashis que se pueden generar a partir del
hashi de partida
     */
     public  ArrayList<Hashi> compleciones(Hashi hashiDePartida){

        ArrayList<Hashi> listaHashisNuevos = new ArrayList<Hashi>();
```

```
      if(hashiDePartida.cumpleCondicionesDePoda() ){
         //recorro la lista de islas candidatas generando ensayos
         for(Isla islaMirada : hashiDePartida.obtenerListaCandidatas()){
            //genero la lista de las vecinas de la isla candidata a las que se esta¡ en condiciones de
tender algun puente
            ArrayList<Isla> listaDeVecinas = hashiDePartida.islasVecinas(islaMirada);
            //voy generando un ensayo por cada isla vecina
            // e introduciendolo en la lista de Hashis a devolver
            for(Isla islaVecina : listaDeVecinas){
               //Hay que evitar la duplicidad de los puentes dobles en la lista de compleciones
               boolean hashiDuplicado = false;
               //Hago una copia local del hashi y las islas que me pasan, las modificaciones sobre
estas copias no
               // afectaran a los originales
               Hashi hashiMirado = hashiDePartida.clonar();
               Isla clon_islaMirada = islaMirada.clonar();
               Isla clon_islaVecina = islaVecina.clonar();
               Hashi hashiEnsayo = generar_ensayo( hashiMirado,clon_islaMirada,clon_islaVecina);
               //Puede que la generacion del ensayo me devuelva un hashi nulo, en ese caso es que el
ensayo
               // ha conducido a un hashi irresoluble y habra que probar con otro
               if(hashiEnsayo == null)
                  continue;

               //Puede que el hashi ensayo sea ya solución en cuyo caso limpio la lista de las
compleciones
               // encontradas hasta ahora, lo meto en ellas (para que solo este la solución)
               //y devuelvo inmediatamente las mismas al vuelta atrás


               if(hashiEnsayo.solucion()){
                  listaHashisNuevos.clear();
                  listaHashisNuevos.add(hashiEnsayo);
                  return listaHashisNuevos;
               }

               //Miro a ver si el hashiEnsayo esta ya en el almacen de Hashis que ya fueron
generados,
               // si esta¡ en el almacen, no lo meto en la lista de compleciones y si no estaba en el
almacen
               // lo meto en el y en la lista de compleciones
              hashiDuplicado = AlmacenDeHashis.hashiDuplicado(hashiEnsayo);
               if(hashiDuplicado){
                  hashiDuplicado = false;
                  //no meto a este hashi duplicado en la lista de Hashis a entregar al Vuelta Atras
                  continue;
               }
               else{
               AlmacenDeHashis.agregarAalmacenHashis( hashiEnsayo);
               listaHashisNuevos.add(hashiEnsayo);
               // para que funcione el metodo generar_ensayo, la islaMirada y la vecina han de estar
```

```
                // dentro de la lista de islas del hashiDePartida
                 }

            }
          }
       }
       //HEURISTICA: ordeno la lista de hashis nuevos según la longitud de su lista de canditatas de menor a
       // mayor, generalmente la accion de poner primero las compleciones mas restrictivas hace más eficiente la vuelta atras (wikipedia)
       listaHashisNuevos = OrdenarHashisSegunCandidatas (listaHashisNuevos);
       return listaHashisNuevos;
}


   /**
    * El algoritmo de vuelta atras
    *
    *  @param   hashi   el hashi sobre el que se va a aplicar el algoritmo
    *  @param   myEntSal datos para poder imprimir la solucion
    *  @param   depuracion si es =1 se activa la depuracion y se imprimen los hashis intermedios
    *           y si es =0 solo se imprime la solucion final
    *
    *  @return acabado  me indica si se ha llegado al final del hashi o no
    */
   public  boolean vueltaAtras(Hashi hashi, EntSal myEntSal, boolean depuracion){


       if (InterfaceUsuario.nodos == 0){
          cota = Tratamiento.hashiInicial.obtenerListaDeIslas().size();
       }

       ++InterfaceUsuario.nodos;
       nodos = InterfaceUsuario.nodos;

       if (nodos == 5)
          acabado = false;

       //si el hashi es solución lo imprimo
       if(hashi.solucion()){
          hashi.imprimirSolucion(myEntSal,Tratamiento.obtenerRutaEntrada());
          acabado = true;
          return true;
          }
       else if (nodos > nodosLimite){
          EntSal miEntSal = new EntSal();
          System.out.println();
          System.out.println();
          System.out.println("Se han alcanzado los nodos maximos en la busqueda de la solucion: "+ (nodos - 1) + " nodos");
          System.out.println("Si quiere seguir buscando la solucion, comience el programa con un valor mayor para el maximo de los nodos ");
```

```
            //Acabo aqui el programa
            System.exit(1);
        }
        else {
            //HEURISTICA: Hago una revision de los avances del algoritmo cada diez nodos: si no se
ha disminuido significativamente el numero
            // de islas (si no ha disminuido la cota) es que no hay avances y entonces recomienzo con el
siguiente hashi de las compleciones del hashi inicial
            // .Si por el contrario si se han producido avances, actualizo el valor de la cota ,
disminuyendolo al numero de islas actual
            if(InterfaceUsuario.nodos == 10 * i){
                if( hashi.obtenerListaDeIslas().size() <= cota){
                    cota = hashi.obtenerListaDeIslas().size();
                    i++;
                     }
                else{
                    hashi = compleciones(Tratamiento.hashiInicial).get(i);
                    i++;
                }
            }

            ArrayList<Hashi> complecionesHashi = compleciones(hashi);

            for(Hashi hashiMirado : complecionesHashi){

                // sentencia necesaria para volver de las llamadas recursivas
                //al metodo pruebaVueltaAtras
                if(acabado)
                    return true;

                // En el modo depuracion imprimo los resultados intemedios
                if(depuracion)
                    hashiMirado.imprimirHashi(myEntSal,InterfaceUsuario.nodos);

                System.out.print(" \t Procesados "+ nodos   + " nodos \r");
                vueltaAtras(hashiMirado, myEntSal,depuracion);

            }
        }

      return acabado;
    }

/**
 * Ordena un ArrayList de Hashis segun el tamaño de sus listas de candidatas
 */
private ArrayList<Hashi> OrdenarHashisSegunCandidatas (ArrayList<Hashi> listaHashis){

    listaHashis = Ordenar_Insercion(listaHashis);

    return listaHashis;
```

```
}

/**
 * Ordena un ArrayList de Hashis segun el tamaño de sus listas de candidatas por el algoritmo de insercion
 */

public static ArrayList<Hashi> Ordenar_Insercion(ArrayList<Hashi> listaHashis){


        for (int i = 1; i < listaHashis.size(); i++) {

            Hashi temp = listaHashis.get(i);
            int j;
            for (j = i- 1; j >= 0 && temp.obtenerListaCandidatas().size() < listaHashis.get(j).obtenerListaCandidatas().size()  ; j--) {

                listaHashis.set(j + 1 , listaHashis.get(j));
            }
            listaHashis.set(j + 1,  temp);

            }
        return listaHashis;
    }

}
```

## - La clase Hashi -

```
import java.util.HashSet;
import java.util.ArrayList;

/**
 * Utilizando los metodos y objetos de las clases Puente e Isla, implemento en esta clase todos los metodos que me
 * sirvan para trabajar con los tableros "Hashis"
 */
public class Hashi
{

   //En los conjuntos de puentes guardo la orientacion, el tipo (doble,simple), las filas que atraviesa cada puente
   //y un par de referencias a las islas que conecta que se guardan  en la listaDeIslas

    private  HashSet<Puente> conjuntoDePuentesVerticales;
    private  HashSet<Puente> conjuntoDePuentesHorizontales;
    private  ArrayList<Isla> listaDeIslas;

    // La lista de candidatas estara compuesta de las islas de la componente conexa
    //  susceptibles de tender un puente
    private  ArrayList<Isla> listaDeCandidatas;

   /**
    * Constructor de la clase
    */
   public Hashi()
   {
      // initialise instance variables
      conjuntoDePuentesVerticales = new HashSet<Puente>();
      conjuntoDePuentesHorizontales= new HashSet<Puente>();
      listaDeIslas = new ArrayList<Isla>();
      listaDeCandidatas = new ArrayList<Isla>();
   }


   /**
    * Constructor de la clase con parametros
    */
   public Hashi(ArrayList<Isla> listaInicial)
   {
      // initialise instance variables
      conjuntoDePuentesVerticales = new HashSet<Puente>();
      conjuntoDePuentesHorizontales= new HashSet<Puente>();
      listaDeIslas = listaInicial;

      // en la lista de candidatas pongo la primera isla de la lista inicial
      listaDeCandidatas = new ArrayList<Isla>();
      if(listaInicial.size() > 0)
         listaDeCandidatas.add(listaInicial.get(0));
```

```
}

/**
 * Constructor de la clase con todos los parametros
 */
public Hashi(ArrayList<Isla> listaDeIslas,ArrayList<Isla> listaCandidatas ,
             HashSet<Puente> conjuntoDePuentesVerticales,
               HashSet<Puente>conjuntoDePuentesHorizontales )
{
   this.conjuntoDePuentesVerticales = conjuntoDePuentesVerticales;
   this.conjuntoDePuentesHorizontales = conjuntoDePuentesHorizontales;
   this.listaDeIslas = listaDeIslas;
   this.listaDeCandidatas = listaCandidatas;
}

/**
 * Estractor de la lista de islas
 */
public ArrayList<Isla> obtenerListaDeIslas(){

   return listaDeIslas;
}

/**
 * Estractor de las candidatas
 */
public ArrayList<Isla> obtenerListaCandidatas(){

   return listaDeCandidatas;
}

/**
 * Estractor de campos de la clase : devuelve los puentes verticales del Hashi
 */

public HashSet<Puente> obtenerPuentesVerticales(){

   return conjuntoDePuentesVerticales;

}

/**
 * Estractor de campos de la clase: devuelve los puentes horizontales del Hashi
 */

public HashSet<Puente> obtenerPuentesHorizontales(){

   return conjuntoDePuentesHorizontales;

}

/**
```

```
    * Clonador de Hashis
    */
   public  Hashi clonar(){

       ArrayList<Isla> listaClonada = new ArrayList<Isla>();
       listaClonada = clonarLista(listaDeIslas);

       ArrayList<Isla> CandidatasClonada = new ArrayList<Isla>();
       CandidatasClonada = clonarLista(listaDeCandidatas);

       HashSet<Puente> conjPuentVertClonados = new HashSet<Puente>();
       conjPuentVertClonados = clonarConjuntoPuentes(conjuntoDePuentesVerticales);

       HashSet<Puente> conjPuentHorClonados = new HashSet<Puente>();
       conjPuentHorClonados = clonarConjuntoPuentes(conjuntoDePuentesHorizontales);

       Hashi hashiClonico = new Hashi(listaClonada,CandidatasClonada,
conjPuentVertClonados,conjPuentHorClonados);

       return hashiClonico;
    }


   /**
    * Clonador de lista de Islas
    */
   static public  ArrayList<Isla> clonarLista(ArrayList<Isla> listaOriginal) {

       ArrayList<Isla> listaClonica = new ArrayList<Isla>();
       for(Isla isla : listaOriginal){

             listaClonica.add(isla.clonar());
       }
       return listaClonica;
   }


   /**
    * Clonador de Conjunto de puentes Verticales
    */
   static public  HashSet<Puente> clonarConjuntoPuentes(HashSet<Puente> conjuntoOriginal) {

       HashSet<Puente> conjuntoClonico = new HashSet<Puente>();

       for(Puente puente: conjuntoOriginal)
          conjuntoClonico.add(puente.clonar());
       return conjuntoClonico;
   }



    /**
```

```
    * Dadas dos islas me dice si se puede tender un puente entre ellas sin que cruce a una isla
    * u otro puente entre ambas
    */

   public boolean sePuedeTender(Isla islaA, Isla islaB){

       return (!(Isla.iguales(islaA,islaB)) &&
               (islaA.islasAlaVista(islaA,islaB, Tratamiento.listaDeIslasMadre)) &&
               (!(Puente.puenteDoble(islaA,islaB,conjuntoDePuentesVerticales)) &&
                !(Puente.puenteDoble(islaA,islaB,conjuntoDePuentesHorizontales))) &&
               (!(Puente.puentesCruzados(islaA,
islaB,conjuntoDePuentesVerticales,conjuntoDePuentesHorizontales))));
    }
    /**
    * Dadas dos islas me dice si existe un puente entre ambas
    */
   public boolean seHaTendido(Isla islaA, Isla islaB){

       boolean conectadas = false;
       Puente puenteVertical = new Puente();
       Puente puenteHorizontal = new Puente();

       puenteVertical =  Puente.mirarPuente(islaA, islaB,conjuntoDePuentesVerticales);
       puenteHorizontal =  Puente.mirarPuente(islaA, islaB,conjuntoDePuentesHorizontales);

       if ((puenteVertical != null) || (puenteHorizontal != null))
          conectadas = true;

       return conectadas;
    }

    /**
    * IslasVecinas me devuelve una lista con las islas vecinas a la isla considerada, con las que se
esta en
    *  condiciones de tender algun puente
    */
    public ArrayList<Isla> islasVecinas (Isla isla){

       ArrayList<Isla> listaDeVecinas = new ArrayList<Isla>();

       boolean vecinaEncontrada = false;

       for(Isla islaMirada : listaDeIslas){

          vecinaEncontrada = false;

          if ( sePuedeTender(islaMirada,isla)){

             vecinaEncontrada = true;
             }

          //HEURISTICA: si la vecina mirada no tiene a su vez vecinas, el puente con la isla original
```

```
es
            // obligado y por tanto , para favorecer cuanto antes su formacion, lo pongo al principio de la
            // lista de vecinas
            if ( vecinaEncontrada && (numeroVecinas(islaMirada) <= 2)){

                listaDeVecinas.add(0,islaMirada);
            }
            else if( vecinaEncontrada ){
                //HEURISTICA: para favorecer la creacion de puentes simples frente a los dobles
                //Si no hay ningun puente previo entre ambas, la isla mirada se pone al
                // principio de la lista de vecinas
                if((Puente.mirarPuente(isla,islaMirada,conjuntoDePuentesVerticales) == null) &&
                    (Puente.mirarPuente(isla,islaMirada,conjuntoDePuentesHorizontales) == null))

                    listaDeVecinas.add(0,islaMirada);
                else
                    //Si hay puente previo se pone al final de la lista de vecinas
                    listaDeVecinas.add(islaMirada);

            }

        }

        return listaDeVecinas;
    }

    /**
     * Me devuelve una lista con las islas que estan conectadas , tienen puentes tendidos , a una isla
dada
     */
    public ArrayList<Isla> conectadas (Isla isla){

        ArrayList<Isla> islasConectadas = new ArrayList<Isla>();

        for(Isla islaMirada : Tratamiento.listaDeIslasMadre){

            if(seHaTendido(isla, islaMirada))
                islasConectadas.add(islaMirada);
        }
        return islasConectadas;
    }

    /**
     * Devuelve verdadero si las TODAS las islas y los puentes del hashi forman un conjunto conexo.
     * para ello uso un recorrido en anchura basado en la estructura de datos de la cola
     */
    public boolean conexo(){

        Isla islaInicial = new Isla();
        Isla islaSiguiente = new Isla();

        int numeroIslasHashi = Tratamiento.listaDeIslasMadre.size() ;
```

```
        int estado[] = new int[numeroIslasHashi ]; //Java inicializa a 0 por defecto todos los elementos de este vector

        Cola cola = new Cola <Isla>();

        //Tomo la primera isla que encuentre que tenga puente y la uso para, a partir de ella ir
        // mirando las islas que estan conectadas
        for(Isla islaMirada : Tratamiento.listaDeIslasMadre){

            if(conectadas(islaMirada).size() != 0){
                islaInicial = islaMirada;
                break;
            }
        }

        int indice = Tratamiento.listaDeIslasMadre.indexOf(islaInicial);
        estado[indice] = 1;
        cola.encolar(islaInicial);

        while (!cola.colaVacia()){
            islaSiguiente = (Isla) cola.desencolar();
            for(Isla islaMirada : conectadas(islaSiguiente)){

                if(estado[Tratamiento.listaDeIslasMadre.indexOf(islaMirada)] == 0){

                    estado[Tratamiento.listaDeIslasMadre.indexOf(islaMirada)] = 1; //marca la isla como visitada
                    cola.encolar(islaMirada);
                }
            }
        }
        //El grafo sera conexo si todas las islas han sido visitadas ,( el estado de cada isla es igual a 1),
        // por lo que la suma del array de estados debe de dar igual al numero de islas
        int sumaEstados = 0;
        for (int i=0; i < estado.length; i++)
            sumaEstados += estado[i];
        boolean conex = (Tratamiento.listaDeIslasMadre.size() == sumaEstados);
        return conex;

    }
    /**
     * Me devuelve una lista con las islas que PUEDEN estar conectadas en un futuro , a una isla dada.
     * Si la isla no esta en la listaDeIslas me devuelve null ya que no puede tender ningun puente
     * potencial al haberlos tendido ya todos los que podia
     */
    public ArrayList<Isla> potencialmenteConectadas (Isla isla){

        ArrayList<Isla> islasPotencialmenteConectadas = new ArrayList<Isla>();

        //Si la isla esta agotada devuelve una lista vacia
        if(!Listas.contiene( isla, listaDeIslas))
```

```
            return islasPotencialmenteConectadas;

        for(Isla islaMirada : Tratamiento.listaDeIslasMadre){

            if(sePuedeTender(isla, islaMirada) && Listas.contiene( islaMirada, listaDeIslas) )
                islasPotencialmenteConectadas.add(islaMirada);
        }
        return islasPotencialmenteConectadas;
    }

    /**
     * Un hashi es potencialmenteConexo cuando las islas que ya no estan en la lista de islas (las islas "Agotadas" que han tendido todos
     * sus puentes) PUEDEN SER  conectadas mediante algún camino a alguna isla que aún está en la lista de Islas.
     * De no ser así se estarian formando regiones inconexas, incompatibles con la solucion correcta del problema.
     * Utilizo un recorrido en anchura basado en la estructura de datos de  cola para implementar este metodo.
     */
    public boolean potencialmenteConexo(){

        boolean loEs = true;
        boolean encontradaAislada = false;
        ArrayList<Isla> islasAgotadas = new ArrayList<Isla>();
        ArrayList<Isla> islasAgotadasConexas = new ArrayList<Isla>();

        //Genero la lista de las islas agotadas
        for(Isla islaMirada : Tratamiento.listaDeIslasMadre){

            if(!Listas.contiene( islaMirada, listaDeIslas))
                islasAgotadas.add(islaMirada);
            }

        //Recorro la lista de las islas agotadas buscando un camino potencial o real entre la isla agotada y
        // una isla por agotar
        ArrayList<Isla> listaConectadas = new ArrayList<Isla>();;
        Isla islaSiguiente = new Isla();
        int numeroIslasHashi = Tratamiento.listaDeIslasMadre.size() ;
        int estado[] = new int[numeroIslasHashi ];
        Cola cola = new Cola <Isla>();

        for(Isla islaAgotada : islasAgotadas){

            loEs = false;
            int indice = Tratamiento.listaDeIslasMadre.indexOf(islaAgotada);
            for (int x=0; x < estado.length; x++)
                estado[x] = 0;
            estado[indice] = 1;
            cola.vaciarCola();
            cola.encolar(islaAgotada);
```

```
        listaConectadas.clear();
        boolean encontradoCamino = false;

        if((Isla.obtenerColum(islaAgotada) == 0) && (Isla.obtenerFila(islaAgotada) == 9))
           loEs = false;

        while (!cola.colaVacia()){
           islaSiguiente = (Isla) cola.desencolar();

           for(Isla isla1 : conectadas(islaSiguiente)){
              listaConectadas.add(isla1);
           }
           for(Isla isla2 : potencialmenteConectadas(islaSiguiente)){
              listaConectadas.add(isla2);
           }
           for(Isla islaMirada2 : listaConectadas){

              if(Listas.contiene( islaMirada2, listaDeIslas)){
                 loEs = true;
                 encontradoCamino = true;
                 break;
              }
              if(estado[Tratamiento.listaDeIslasMadre.indexOf(islaMirada2)] == 0){

                 estado[Tratamiento.listaDeIslasMadre.indexOf(islaMirada2)] = 1; //marca la isla
como visitada
                 cola.encolar(islaMirada2);
              }
           }
           if(encontradoCamino){
              break;
           }
        }
        if( !encontradoCamino)
           return false;
        }

     return loEs;
   }

   /**
    * Me da el numero de vecinas de una isla
    */
   public int numeroVecinas (Isla isla){

     int numero = 0;

     for(Isla islaMirada : listaDeIslas){

        if( sePuedeTender(islaMirada,isla))
           ++numero;
```

```
        }
        return numero;
    }

    /**
     * Puentes posibles me devuelve el numero de puentes que aun se pueden tender desde una isla a
sus vecinos
     */
    public int puentesPosibles(Isla isla){

        int posibles = 0;
        //lista de vecinas a las que puedo tender algÃºn puente
        ArrayList<Isla> listaDeVecinas = islasVecinas(isla);

        for(Isla vecinaMirada : listaDeVecinas){

          //Si ya hay puente previo, o la isla vecina o la mirada valen 1, solo es posible un puente mÃ¡s
entre
          // estas dos islas
          if((obtenerValorEnLista(isla) == 1) || (obtenerValorEnLista(vecinaMirada) == 1) ||
             (Puente.mirarPuente(isla,vecinaMirada,conjuntoDePuentesVerticales) != null) ||
             (Puente.mirarPuente(isla,vecinaMirada,conjuntoDePuentesHorizontales) != null))

              posibles++;

          else
              //Si no hay puente previo y la isla vecina y la mirada valen mas de dos. se pueden tener
              // dos puentes
              posibles = posibles + 2;
           }


        return posibles;
    }

    /**
     * Toma dos islas de la lista de islas , las enlaza  disminuyendo
     * en uno el valor guardado en A y en B , crea un puente entre ambas y lo mete en el conjunto de
     * los puentes, y quita a una o ambas islas de la lista de las islas por enlazar si sus valores
     * llegan a cero
     */
    public void tenderPuente(Isla islaA, Isla islaB)
    {
        //quito temporalmente a las islas del puente que se va a tender de la lista de candidatas
        for(Isla candidataMirada : listaDeCandidatas){

           if(Isla.iguales(islaA, candidataMirada)){
              listaDeCandidatas.remove(candidataMirada);
              break;
           }
        }
        for(Isla candidataMirada : listaDeCandidatas){
```

```
            if(Isla.iguales(islaB, candidataMirada)){
                listaDeCandidatas.remove(candidataMirada);
                break;
            }
        }
        //Actualizo los conjuntos de puentes
        if(Isla.obtenerColum(islaA) == Isla.obtenerColum(islaB))
            Puente.actualizarPuentes(islaA,islaB,conjuntoDePuentesVerticales);
        if(Isla.obtenerFila(islaA) == Isla.obtenerFila(islaB))
            Puente.actualizarPuentes(islaA,islaB,conjuntoDePuentesHorizontales);

        //actualizo la situacion de las islas en la lista,disminuyo en uno el valor de las islas que
componen
        // el nuevo puente si, tras tender el puente,
        //el valor de estas ha llegado a 0 , tengo que quitarlas de la lista, ademas si su
        // valor es distinto de 0 tengo que volver a hacerla isla candidata
        actualizarListaDeIslas(islaA);
        actualizarListaDeIslas(islaB);
       // HEURISTICA para evitar la rapida formacion de puentes dobles:
       // si alguna de estas islas solo puede tender puentes dobles (tiene ya puentes con todas
       // sus vecinas ) la pongo al final de la lista de candidatas, recolocando la lista de candidatas
       recolocaCandidatas();
    }

    /**
     * Actualizo la situacion de las islas en la lista, si, tras tender el puente,
     * el valor de estas ha llegado a 0 , tengo que quitarlas de la lista , ademas si su
     * valor es distinto de 0 tengo que volver a hacerla isla candidata
     */
   public void actualizarListaDeIslas(Isla isla) {

         // Busco en la lista de islas aquella que coincida en valores con la isla a
         // actualizar
         for(Isla islaMirada : listaDeIslas){

             if(Isla.iguales(isla, islaMirada)){

                 if(obtenerValorEnLista(islaMirada) == 1) {
                     //si la isla, tras la disminucion de valor llegara a 0 la quito de la lista de islas
                     // y de la lista de candidatas
                     listaDeIslas.remove(islaMirada);
                     Listas.borra(islaMirada, listaDeCandidatas);
                     break;

                   }
                 else{
                   // sin no, le disminuyo el valor en la lista
                  disminuyeValorEnLista(islaMirada);
                  // y la coloco en la lista de candidatas
                  listaDeCandidatas.add(isla);
                   }
```

```
            }
          }
        }
 /**
  * HEURISTICA para evitar la rapida formacion de puentes dobles:
  * voy revisando la lista de las candidatas y si alguna de estas islas
  * solo puede tender puentes dobles (tiene ya puentes con todas
  * sus vecinas ) la pongo al final de la lista de candidatas
  */
 public void recolocaCandidatas(){

    boolean tieneUnoLibre = false;
    ArrayList<Isla> listaDeVecinas;
    int [ ] indiceCandidataArecolocar = new int[25*25 ];

    int i = 0;

    //primero reviso la lista de candidatas y voy tomando nota en un array de indices de las que
luego tengo
    // que recolocar
    for(Isla candidataMirada : listaDeCandidatas){

        tieneUnoLibre = false;
        //primero genero la lista de vecinas de la isla
        listaDeVecinas = islasVecinas (candidataMirada);

        //si encuentro alguna vecina con la que no tenga tendido ningÃºn puente todavia, no
tendrÃ©
        // que recolocar esta isla en las candidatas ya que puede tender aÃºn al menos un puente
        // simple
        for(Isla vecinaMirada : listaDeVecinas){

            if((Puente.mirarPuente(candidataMirada,vecinaMirada,conjuntoDePuentesVerticales) ==
null) &&
               (Puente.mirarPuente(candidataMirada,vecinaMirada,conjuntoDePuentesHorizontales)
== null)){

                  tieneUnoLibre = true;
                  break;
               }
            }

        if (!tieneUnoLibre){
              indiceCandidataArecolocar[i] = listaDeCandidatas.indexOf(candidataMirada) ;
              i++;
            }
        }
    // si he encontrado que es una isla que va a dar solo puentes dobles, la recoloco al final
    // de la lista de candidatas
    int j ;
    for(i = 0 ; i <= indiceCandidataArecolocar.length - 1 ; i++){
```

```
            if(indiceCandidataArecolocar[i] == 0 && indiceCandidataArecolocar[i + 1] == 0)
                break;


            Isla candidataArecolocar = listaDeCandidatas.get(indiceCandidataArecolocar[i]);
            listaDeCandidatas.remove(listaDeCandidatas.get(indiceCandidataArecolocar[i]));
            listaDeCandidatas.add(candidataArecolocar);
            //disminuyo en uno los indices de las candidatas a recolocar a partir de i, para compensar el
            // hueco creado por la recolocacion
            j = i + 1;
            while(indiceCandidataArecolocar[j] != 0){

                indiceCandidataArecolocar[j]--;
                j++;
            }
        }
    }
    /**
     * Devuelve la lista de islas adyacentes a un puente
     */
    public ArrayList<Isla> islasAdyacentesAlPuente(Puente puente){

        ArrayList<Isla> adyacentes = new ArrayList();

        adyacentes.add(Puente.obtenerIslaA(puente));
        adyacentes.add(Puente.obtenerIslaB(puente));

        adyacentes.addAll(islasVecinas(Puente.obtenerIslaA(puente)));
        adyacentes.addAll(islasVecinas(Puente.obtenerIslaB(puente)));

        if(Puente.obtenerOrientacion(puente) == Orientacion.vertical){

            int [ ]filas = Puente.obtenerFilas (puente);

            for(Isla islaMirada : listaDeIslas){

                if((!adyacentes.contains(islaMirada)) &&
                    ((Isla.obtenerFila(islaMirada)) >= filas[0]) && (Isla.obtenerFila(islaMirada)) <=
filas[filas.length - 1]){

                    adyacentes.add(islaMirada);
                }
            }
        }

        if(Puente.obtenerOrientacion(puente) == Orientacion.horizontal){

            int [ ]columnas = Puente.obtenerColumnas(puente);

            for(Isla islaMirada : listaDeIslas){

                if((!adyacentes.contains(islaMirada)) &&
```

```
                ((Isla.obtenerColum(islaMirada)) >= columnas[0]) &&
(Isla.obtenerColum(islaMirada)) <= columnas[columnas.length - 1]){

                    adyacentes.add(islaMirada);
                }
            }
        }
        return adyacentes;

    }



 /**
  * Entiendo por "Isla Afortuanada" a aquella a la que le es indiferente la direccion que tome su
  * proximo puente.
  * Esta rodeada por una serie de vecinas y  a todas  tiene que tender al menos
  * un puente.
  * Este es un metodo heuristico que devuelve verdadero si la isla es afortunada y esta forzada a
tender una serie
  * de puentes. P.ej. una isla de valor 8 no tiene eleccion sobre los puentes a tender, igual que
  * una isla de valor 4 en una esquina , una de valor 6 en la pared etc...
  */
 public boolean islaAfortunada(Isla isla){

     boolean afortunada = false;
     ArrayList<Isla> listaVecinas = islasVecinas (isla);

     int numeroDeVecinas = listaVecinas.size();
     int valorIsla = obtenerValorEnLista(isla);

      //Desglose por casos de la aparicion de una isla afortunada

      // isla afortunada de valor par
      if( (( valorIsla == 2) && (numeroDeVecinas == 1 ))  ||
          (( valorIsla == 4) && (numeroDeVecinas == 2 ))  ||
          (( valorIsla == 6) && (numeroDeVecinas == 3 ))  ||
          (( valorIsla == 8) && (numeroDeVecinas == 4 ))   )
          {
       afortunada = true;
       return afortunada;
      }

      //isla afortunada de valor impar
      if( (( valorIsla == 1) && (numeroDeVecinas == 1 ))  ||
          (( valorIsla == 3) && (numeroDeVecinas == 2 ) && (vecinaUnitaria(isla)))  ||
          (( valorIsla == 5) && (numeroDeVecinas == 3 ) && (vecinaUnitaria(isla)))  ||
          (( valorIsla == 7) && (numeroDeVecinas == 4 ) && (vecinaUnitaria(isla)))    )
         {
       afortunada = true;
       return afortunada;
      }
```

```
    return afortunada;
   }

 /**
  * Entiendo por "Isla semi Afortuanada" a aquella a la que le es indiferente la direcciones que tomen sus
  *  puentes, menos el último , que queda indefinido.
  *  Se da en el caso de islas con valor impar
  *  P.ej. una isla de valor 3 rodeada por dos vecinas, una de valor 5 rodeada por 3 vecinas y una de valor
  *  7 rodeada por 4 vecinas
  */
 public boolean islaSemiAfortunada(Isla isla){

    boolean SemiAfortunada = false;
    ArrayList<Isla> listaVecinas = islasVecinas (isla);

    int numeroDeVecinas = listaVecinas.size();
    int valorIsla = obtenerValorEnLista(isla);

    if( (( valorIsla == 3) && (numeroDeVecinas == 2 ))  ||
         (( valorIsla == 5) && (numeroDeVecinas == 3 ) )  ||
         (( valorIsla == 7) && (numeroDeVecinas == 4 ) ))
        {
       SemiAfortunada = true;
       return SemiAfortunada;
      }
    return SemiAfortunada;
   }

  /**
   * Un Hashi sera parcialmente valido cuando:
   *
   *   El valor de una isla (el num de puentes a tender a partir de una isla) sea menor o igual
   *   al num de puentes posibles con sus vecinas a partir de esa isla en el momento actual.
   *
   *   Con esto evito situaciones del tipo : (2)--(2) y (1)--(3)--(1) p.ej. donde el número de puentes
   *   que se pueden tender es menor al necesario para dejar a cero los valores de las islas.
   *
   */
  public boolean parcialmenteValido(){

      boolean valido = true;

       for(Isla islaVista : listaDeIslas){

          if(obtenerValorEnLista(islaVista) > puentesPosibles(islaVista)){
             valido = false;
             break;
             }
         }
```

```
    return valido;
}

/**
 * CONDICIONES DE PODA
 * El metodo me dara verdadero si el ensayo hashi
 * al que se lo paso cumple las condiciones de poda
 */
public boolean cumpleCondicionesDePoda(){

    boolean lasCumple = true;

    if(!potencialmenteConexo())
        return false;

   if(!parcialmenteValido())
        return false;

    return lasCumple;

}
/**
 * Devuelve verdadero si la isla tiene una vecina de valor 1
 */
public boolean vecinaUnitaria(Isla isla){

    boolean unitaria = false;
    ArrayList<Isla> listaVecinas = islasVecinas (isla);

    for(Isla islaVecina : listaVecinas){

                if(obtenerValorEnLista(islaVecina) == 1 ) {
                    unitaria = true;
                    break;
                }
            }
    return unitaria;
}



 /**
 * Obtiene el valor de una isla de coordenadas determinadas en la lista de islas
 */
public int obtenerValorEnLista(Isla isla){

    int valor = 0;

    for(Isla islaMirada : listaDeIslas){

        if(Isla.iguales(islaMirada, isla)){
```

```
            valor = islaMirada.obtenerValor();
            break;

        }
    }
    return valor;
}

/**
 * Disminuye en uno el  valor de una isla de coordenadas determinadas en la lista de islas
 */
public void disminuyeValorEnLista(Isla isla){

    int valor;

    for(Isla islaMirada : listaDeIslas){

        if(Isla.iguales(islaMirada, isla)){

            valor = islaMirada.obtenerValor();
            islaMirada.ponerValor(valor - 1);
            break;

        }
    }
}



/**
 * Dada una isla, busca a su homologa en  la lista del hashi
 */
public Isla buscaIsla (Isla islaBuscada){

    Isla islaEncontrada = null;

    for(Isla islaMirada : listaDeIslas){

        if( Isla.iguales(islaBuscada, islaMirada)) {

            islaEncontrada = islaMirada;
            break;
        }
    }

    return islaEncontrada;
}

 /**
 * Imprime una posible solucion, (puede que sea parcial)
 */
```

```
    public void imprimirHashi(EntSal myEntSal, int nodo){


myEntSal.imprimirHashi(conjuntoDePuentesVerticales,conjuntoDePuentesHorizontales,listaDeIsla
s);

    }

    /**
     * Imprime la solucion final
     */
    public void imprimirSolucion(EntSal myEntSal,String rutaEntradaDatos){


myEntSal.imprimirHashi(conjuntoDePuentesVerticales,conjuntoDePuentesHorizontales,Tratamient
o.listaDeIslasMadre);

    }

    /**
     * Devuelve verdadero si el Hashi representa una solucion valida para el tablero inicial
     */
    public boolean solucion(){

        return listaDeIslas.isEmpty() && conexo();
    }


}
```

## - La clase Tratamiento -

```
/**
 * Esta clase se encarga de tratar los  Hashis realizando sucesivas pasasadas en busca de islas afortunadas y
 * semi afortunadas para tender los puentes de las mismas y devolver al hashi ya tratado.
 *
 * Esta clase se utiliza en dos contextos:
 * 1- Con el método "generarHashiInicial" ,al iniciar el programa para realizar
 * el pretratamiento del tablero inicial en busca de afortunadas antes de pasarselo al algoritmo de vuelta atras ,
 * esto puede facilitarle mucho las cosas al vuelta atras pues puede que este tratamiento simplifique sobremanera el tablero inicial.
 *
 * 2- Dentro del algoritmo vuelta atras cuando se genera un ensayo tendiendo un puente entre las islas se llama seguidamente al
 * metodo "tratarHashi"  para que compruebe si el nuevo puente ha provocado la creación de nuevas afortunadas o semiafortunadas.
 *
 */
import  java.io.*;
import  java.util.*;

public class Tratamiento
{
  static private String rutaEntradaDatos ;
  static public EntSal EntSalDatos;

  static public  ArrayList<Isla> listaDeIslasMadre;

  static public  ArrayList<Isla> listaDeIslasIniciales;
  static public  ArrayList<Isla> listaOriginal;
  static public Hashi hashiInicial;
  static public Vuelta_Atras vueltaAtras;


   /**
    * Constructor de la clase Tratamiento
    */
   public Tratamiento(String rutaEntradaDatos, boolean depuracion)
   {
      this.rutaEntradaDatos = rutaEntradaDatos;

      EntSalDatos= new EntSal();
      listaDeIslasMadre = EntSalDatos.creaListaDeIslas(rutaEntradaDatos);
      listaDeIslasIniciales = clonarLista(listaDeIslasMadre);
      listaOriginal = clonarLista(listaDeIslasMadre);
      hashiInicial = generarHashiInicial(listaDeIslasIniciales, depuracion);
```

```
    }

    /**
     * Devuelve la ruta de entrada de los datos
     */
    public static String obtenerRutaEntrada()
    {
        return rutaEntradaDatos;
    }

    /**
     * Metodo para generar el Hashi inicial partiendo de la lista inicial de islas
     * y tendiendo los puentes de las islas afortunadas y semiafortunadas en sucesivas pasadas
     *
     * El metodo devuelve null si los datos iniciales no representan a un hashi valido
     */
    public static Hashi generarHashiInicial(ArrayList<Isla> listaDeIslasIniciales, boolean
depuracion){

        if(!inicialmenteValido(listaDeIslasIniciales))
            return null;

        EntSal myEntSal = new EntSal();
        Hashi hashiOriginal = new Hashi(listaOriginal);
        Hashi hashiInicio = new Hashi(listaDeIslasIniciales);
        boolean nuevasAfortunadas = false;

        int pasada = 0;
        hashiInicio = tratarHashi(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada, myEntSal,
depuracion);

        return hashiInicio;

    }

    /**
     * Al tratar el Hashi voy tendiendo los puentes de las islas afortunadas y semi afortunadas en
sucesivas pasadas, ya que con una
     * sola pasada no basta , porque el tender un puente puede hacer que las islas adyacentes se
conviertan en afortunadas y al tender
     * los puentes de estas se formen nuevas afortunadas, creando así una cascada de afortunadas.
     */
    static public Hashi tratarHashi(Hashi hashiInicio, Hashi hashiOriginal,boolean
nuevasAfortunadas, int pasada, EntSal myEntSal, boolean depuracion){


        hashiInicio = buscaAfortunadas(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada,
myEntSal, depuracion);
        if (hashiInicio == null)
            return null;
        hashiInicio = buscaSemiAfortunadas(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada,
myEntSal, depuracion);
```

```
       if (hashiInicio == null)
           return null;

       // Hago sucesivas pasadas buscando las nuevas afortunadas que se pueden haber
       // creado tras la primera pasada
       do{
           nuevasAfortunadas = false;
           listaOriginal = clonarLista(hashiInicio.obtenerListaDeIslas());
           hashiOriginal = hashiInicio.clonar();

           hashiInicio = buscaAfortunadas(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada,
myEntSal, depuracion);
           if (hashiInicio == null)
               return null;
           hashiInicio = buscaSemiAfortunadas(hashiInicio,hashiOriginal,nuevasAfortunadas, pasada,
myEntSal, depuracion);
           if (hashiInicio == null)
               return null;

        }while(nuevasAfortunadas);

       return hashiInicio;
    }

    /**
     * Busca afortunadas:
     *
     * Busca islas afortunadas en el Hashi y va tendiendo sus islas, devuelve el hashi con todos los
puentes
     * de las islas afortunadas tendidos
     * @param hashiInicio se le pasa este hashi para que busque islas afortunadas en el
     * @return hashiInicio devuelve el hashi con los puentes de las islas afortunadas tendidos
     * @return nuevasAfortunadas pone a verdadera la variable del metodo llamador
"nuevasAfortunadas" si ha
     * encontrado afortunadas nuevas
     *
     */
    static private Hashi buscaAfortunadas(Hashi hashiInicio,Hashi hashiOriginal,boolean
nuevasAfortunadas, int pasada, EntSal myEntSal, boolean depuracion){

       int islasPretratadas = 0;
       for(Isla islaMirada : listaOriginal){

            if(hashiOriginal.islaAfortunada(islaMirada) ){

               int valorInicio = hashiInicio.obtenerValorEnLista(islaMirada);
               int valorOriginal = hashiOriginal.obtenerValorEnLista(islaMirada);
               if (valorInicio == valorOriginal) {
                   //Se imprime un pequeño mensaje en pantalla que informe que el programa esta
progresando
                   System.out.print("\\\\\\ \r");
                   Isla isla = hashiInicio.buscaIsla (islaMirada);
```

```
                    nuevasAfortunadas = true;

                    do{
                       hashiInicio.tenderPuente(isla, hashiInicio.islasVecinas(isla).get(0));
                       valorInicio --;

                       if(!hashiInicio.parcialmenteValido()){
                          hashiInicio = null;
                          return hashiInicio;}

                    }while (valorInicio > 0);

                 if (depuracion)
                    hashiInicio.imprimirHashi(EntSalDatos, -(pasada++));
                 }
              }
           }

        return hashiInicio;

        }

     /**
      * Busca SemiAfortunadas:
      *
      * Busca islas Semi afortunadas en el Hashi y va tendiendo sus islas, devuelve el hashi con todos
los puentes
      * de las islas Semi afortunadas tendidos.
      *
      * Las islas semi afortunadas han de tender un solo puente a cada una de sus vecinas
      *
      * @param hashiInicio se le pasa este hashi para que busque islas afortunadas en el
      * @return hashiInicio devuelve el hashi con los puentes de las islas afortunadas tendidos
      * @return nuevasAfortunadas pone a verdadera la variable del metodo llamador
"nuevasAfortunadas" si ha
      * encontrado afortunadas nuevas
      *
      */
     static private Hashi buscaSemiAfortunadas(Hashi hashiInicio,Hashi hashiOriginal,boolean
nuevasAfortunadas, int pasada, EntSal myEntSal, boolean depuracion){

        int islasPretratadas = 0;
        for(Isla islaMirada : listaOriginal){

              if(hashiOriginal.islaSemiAfortunada(islaMirada) ){

                 int valorInicio = hashiInicio.obtenerValorEnLista(islaMirada);
                 int valorOriginal = hashiOriginal.obtenerValorEnLista(islaMirada);
                 ArrayList<Isla> vecinas = hashiInicio.islasVecinas(islaMirada);

                 if (valorInicio == valorOriginal) {
                    //Se imprime un pequeño mensaje en pantalla que informe que el programa esta
```

```
progresando
                    System.out.print("/// \r");
                    Isla isla = hashiInicio.buscaIsla (islaMirada);
                    nuevasAfortunadas = true;

                    do{
                       hashiInicio.tenderPuente(isla, vecinas.get(0));
                       vecinas.remove(0);
                       valorInicio --;

                       if(!hashiInicio.parcialmenteValido()){
                          hashiInicio = null;
                          return hashiInicio;}

                    }while ((valorInicio > 1 && valorOriginal == 3) ||
                          (valorInicio > 2 && valorOriginal == 5) ||
                          (valorInicio > 3 && valorOriginal == 7));

                if (depuracion)
                   hashiInicio.imprimirHashi(EntSalDatos, -(pasada++));
                }
             }
          }
       return hashiInicio;

       }

   /**
    * Para que un hashi sea inicialmente valido el valor acumulado de sus islas en la lista ha de ser
par
    */
    public static boolean inicialmenteValido(ArrayList<Isla> listaDeIslasIniciales){

       int sumaValores = 0;

       for(Isla islaMirada : listaOriginal){

          sumaValores += islaMirada.obtenerValor();

       }

        return (sumaValores % 2 == 0);

       }



   /**
    * Clonador de lista de Islas
    */
   static public  ArrayList<Isla> clonarLista(ArrayList<Isla> listaOriginal) {
```

```
        ArrayList<Isla> listaClonica = new ArrayList<Isla>();
        for(Isla isla : listaOriginal){

                listaClonica.add(isla.clonar());
        }
        return listaClonica;
    }
}
```

## - La clase almacén de Hashis -

```
import  java.util.*;
/**
 * En el almacen de hashis meto todos los hashis que ya han sido considerados para ir consultandolo
cada vez
 * que genero un hashi nuevo y ver si este es repetido o no
 */
public class AlmacenDeHashis
{

   private static  ArrayList<Hashi> almacenHashis;

    /**
     * Constructor
     */
    public AlmacenDeHashis()
    {
        almacenHashis = new ArrayList<Hashi>();
    }

    /**
     * Agrega hashis al almacen
     */
    static public void agregarAalmacenHashis(Hashi hashi){

        almacenHashis.add(hashi);
    }

    /**
     * Comprueba que un hashiEnsayo no esté duplicado en el Almacen de Hashis
     *
     *  @param  hashi  el hashi  que vamos a estudiar
     * @param  almacen la lista donde estan guardados los hashis a comprobar
     * @return devuelve verdadero si el Hashi está ya en el almacen de Hashis
     * (recordermos que en el almacen de hashis meto todos los hashis que ya han sido considerados ,
     * para evitar considerarlos mas de una vez)
     * */
    static public boolean hashiDuplicado(Hashi hashi){

        boolean hashiRepetido = false;
```

```
        for(Hashi hashiMirado : almacenHashis){

            if(iguales(hashiMirado,hashi)){
                hashiRepetido = true;
                break;
               }
            }
           return hashiRepetido;
}

/**
 * Devuelve verdadero si dos hashis son iguales
 */
  static public boolean iguales(Hashi hashiA, Hashi hashiB){

      boolean igualdadListaIslas  ;
      boolean igualdadConjPuentesVert;
      boolean igualdadConjPuentesHori;

      igualdadListaIslas =
Listas.igualdadListas(hashiA.obtenerListaDeIslas(),hashiB.obtenerListaDeIslas());

      igualdadConjPuentesVert = igualdadConjuntos(hashiA.obtenerPuentesVerticales(),
hashiB.obtenerPuentesVerticales());
      igualdadConjPuentesHori =   igualdadConjuntos(hashiA.obtenerPuentesHorizontales(),
hashiB.obtenerPuentesHorizontales());

      return igualdadListaIslas && igualdadConjPuentesVert && igualdadConjPuentesHori;

    }

   /**
    * Compara dos ArrayList de islas y devuelve verdadero si ambos tienen las mismas islas
    */
   static public boolean igualdadConjuntos (HashSet<Puente> listaA, HashSet<Puente> listaB){

       boolean iguales = true;
       int tamanoA = listaA.size();
       int tamanoB = listaB.size();

       if (tamanoA != tamanoB)
          return false;

       Puente [ ] arrayA = new Puente [listaA.size()];
       Puente [ ] arrayB = new Puente [listaB.size()];

       arrayA  = listaA.toArray(arrayA);
       arrayB  = listaB.toArray(arrayB);

       for (int i = 0; i < tamanoA; i++){
```

```
            if(!(Puente.iguales(arrayA[i],arrayB[i]))) {

                iguales = false;
                break;
            }
        }

        return iguales;

    }

}
```

## - La clase Interface Usuario -

```
/**
 * Esta clase es la que recibe las opciones de la clase hashiwokakero y pone a funcionar el programa con las mismas
 *
 *  entradaDatos: ruta del archivo con el hashi inicial
 *
 *  traza:  es el modo de salida de los datos, si traza = 1 se mostraran los hashis intermedios
 *          y si traza = 0 solo se mostrara el resultado final
 *
 *  nodosLimite: si el programa alcanza ese numero de nodos en su funcionamiento considerará que está tardando demasiado e imprimirá un
 *          mensaje solicitando al usuario permiso para continuar o para empezar con otros parametros
 */
import  java.io.*;
import  java.util.*;

public class InterfaceUsuario
{
    private long t1, t2;
    private float dif;
    private boolean exito;
    public static int nodos;

    private String entradaDatos;
    private boolean traza;
    private int nodosLimite;
    private boolean ayudaAmostrar;

    /**
     * Constructor
     */
    public InterfaceUsuario()
    {
        nodos = 0;
        nodosLimite = 2500;
        traza = false;
        ayudaAmostrar = false;

    }
```

```
/**
 * Actualiza el campo entrada de datos
 */
 public void setNombreFichero(String nombreFichero){

     entradaDatos = nombreFichero;

    }
/**
 * Actualiza el campo traza, poniendo el objeto interfaz en modo traza
 */
 public void setModoTraza(){

     traza = true;

    }

/**
 * Actualiza el campo nodos limite
 */
 public void setNodosLimite(int limite){

     nodosLimite = limite;

    }

/**
 * Actualiza el campo ayudaAmostrar
 */
public void mostrarAyuda(){

    ayudaAmostrar = true;
}
/**
 * Muestra el campo ayudaAmostrar
 */
public boolean getModoAyuda(){

    return ayudaAmostrar;
}

/**
 * Imprime la ayuda
 */
public void ayuda(){

    System.out.println();
    System.out.println();
     String[] str={

     "Programa HASIWOKAKERO",
```

```
"Desarrollado por Diego J. Martinez Garcia para la practica de",
"Programacion III del curso 2009/2010 de Informatica de Sistemas, ",
"",
"   DNI:        45581069z",
"   Lenguaje:   Java",
"   IDE usado:  BlueJ",
"   S.O:        Ubuntu y Windows Vista",
"",
"",
"Este programa puede ejecutarse con las siguientes sintaxis:",
"",
"   hashiwokakero [-t][-h] [fichero]",
"",
"o bien",
"",
"   hashiwokakero -l numero fichero",
"",
"o bien",
"",
"   hashiwokakero -b",
"",
"ejemplos: 'hashiwokakero -t  fichero.txt' ",
"          'hashiwokakero -l 1000 fichero.txt' ",
"          'hashiwokakero -h' ",
"          'hashiwokakero -b' ",
"",
"Opciones:",
"",
"t: Modo traza. Muestra toda la secuencia de resolucion del hashi",
"   mostrando los tableros intermedios .",
"",
"b: Modo bateria de pruebas, realiza la resolucion de 40 hashis",
"   uno tras otro, de manera continua e imprime un resumen de los tiempos .",
"",
"l: Modo limite. si se activa este modo el siguiente argumento ha de ",
"    ser un numero que marcara el limite de nodos de la vuelta atras",
"    el valor por defecto de l es de 2500 nodos                    ",
"",
"h: Modo ayuda. Muestra la sintaxis y los creditos.",
"",
"Si no tiene argumentos de seleccion de modo, el programa muestra la",
" matriz final resuelta, o un mensaje si no la encuentra.",
"",
"Ello permite tambien usar redirecciones y 'pipes' como entrada de",
"datos.",
"",
" ejemplos:",
"     'java hashiwokakero < ejemploHashiOficial.txt' ",
"     'type ejemploHashiOficial.txt | java hashiwokakero'",
"",
"",
```

```
        "   ejemplo de un hashiwokakero: ",
        "",
        "#------------------------",
        "# Ejemplo hashiwokakero",
        "# Fuente: Nikoli",
        "# Valores de la Figura 1",
        "# ----------------------",
        "##",
        "13",
        "13",
        "2 * 3 * * * 4 * * * 2 * *",
        "* * * * * * * * * * * * *",
        "* * * * * * * * * * * 2",
        "* * * * * * * * * * * *",
        "1 * 1 * * * * * 1 * 3 * 3",
        "* * * * * * * * * * * * *",
        "2 * * * * * 8 * * * 5 * 2",
        "* * * * * * * * * * * * *",
        "3 * * * 3 * * * * * * * 1",
        "* * * * * * * * * * * * *",
        "* * * * 2 * * * * * 3 * 4",
        "* * * * * * * * * * * * *",
        "3 * * * * * 3 * 1 * * * 2",
        "+",};
      for (int i=0;i<str.length;i++) System.out.println(str[i]);
    }

    /**
     * Metodo de inicio , que pone a trabajar  a las demas clases
     *
     * Valores iniciales:
     * String entradaDatos = "C:/Users/Diego/Desktop/Practica Programacion 3/PracticaP3/banco de pruebas/ejemploHashi13_13.txt";
     *
     */
    public void IniciarPrograma()
    {
      Calendar ahora1 = Calendar.getInstance();
      t1 = ahora1.getTimeInMillis();
      EntSal myEntSal = new EntSal();

      boolean malConstruido = false;

      System.out.println();
      System.out.println("Pretratando el tablero inicial ...");

      AlmacenDeHashis almacenHashis = new AlmacenDeHashis();

      Vuelta_Atras AlgoritmoVueltaAtras = new Vuelta_Atras(nodosLimite);

      Tratamiento datosPretratados = new Tratamiento(entradaDatos,traza );
```

```
        Hashi hashiInicial = datosPretratados.hashiInicial;

        /* Puede que el Tratamiento halla encontrado que el hashi no tiene solucion, en cuyo caso
tendremos null en el hashi inicial */
        if(hashiInicial == null){
            exito = false;
            malConstruido = true;}
        /* Puede que el Tratamiento halla resuelto el hashi por si mismo */
        else if(hashiInicial.solucion()){
            hashiInicial.imprimirSolucion(Tratamiento.EntSalDatos,Tratamiento.obtenerRutaEntrada());
            exito = true;
        }
        else{
            System.out.println();
            System.out.println("Procesando nodos con el algoritmo de vuelta atras ...");
            exito = AlgoritmoVueltaAtras.vueltaAtras(hashiInicial, Tratamiento.EntSalDatos, traza);
        }
        Calendar ahora2 = Calendar.getInstance();
        t2 = ahora2.getTimeInMillis();

        dif =(float) (t2 - t1) / 1000;

        if(exito){
            System.out.println();
            System.out.println();
            System.out.println("Existe solucion que conecta todas las islas , \n solucion encontrada en  "
+ dif + " segundos.");}
        else if (malConstruido){
            System.out.println();
            System.out.println();
            System.out.println("No existe solucion ya que el hashi esta mal construido. ");}
        else if ( AlgoritmoVueltaAtras.obtenerNodos() <
AlgoritmoVueltaAtras.obtenerNodosLimite()){
            System.out.println();
            System.out.println();
            System.out.println("El hashi no tiene solucion conexa, se ha explorado todo el arbol del
juego \n sin encontrala. ");
        }
        else{
            System.out.println();
            System.out.println();
            System.out.println("No se ha encontrado la solucion para el hashi en " + dif + " segundos
con "+ nodos+ " nodos \n puede que el hashi no tenga solucion o que no se hallan explorado
suficientes nodos \n si quiere hacer una exploración mas profunda comience el programa con un
número mayor para los nodos limite.");
        }
        }
    }
```

## - La clase Hashiwokakero -

```
import java.io.*;

/**
 * Programa que resuelve puzzles hashiwokakeros utilizando el algoritmo de la vuelta atras
 *
 * @author Diego J. Martinez Garcia
 * @version 21/7/2010
 */
public class hashiwokakero
{

   private static String rutaEntradaDatos ;

   /**
    * Constructor
    */
   public hashiwokakero()
   {
      rutaEntradaDatos = new String();
   }

   /**
    * Metodo de arranque del programa
    */
   public static void main(String[] argumentos)throws Exception {

       InterfaceUsuario interfaz = new InterfaceUsuario();
       String nombreFichero;
       boolean entradadesdefichero = false;
       boolean traza = false;
       for (int i=0 ; i<argumentos.length ; i++){
          if (argumentos[i].charAt(0) == '-' ) {
             for (int j=1; j<argumentos[i].length();j++ ){
                switch (argumentos[i].charAt(j)){
                   case 'b':
                   case 'B':
                      BancoDePruebas.testBateriaDePruebas();
                      break;
                   case 'l':
                   case 'L':
                      interfaz.setNodosLimite(Integer.parseInt(argumentos[i + 1]));
                      break;
                   case 't':
                   case 'T':
                      interfaz.setModoTraza();
                      break;
```

```
                    case 'h':
                    case 'H':
                    case '?':
                        interfaz.mostrarAyuda();
                        break;

                    default:
                        System.out.println();
                        System.out.println("error en la linea de comandos:");
                        System.out.println("   parámetro "
                                +argumentos[i].charAt(j)
                                +" desconocido");
                }
            }
        }
        else{
            if (!entradadesdefichero) {
                java.io.File prueba = new java.io.File(argumentos[i]);
                if (prueba.exists()){
                  interfaz.setNombreFichero(argumentos[i]);
                  entradadesdefichero=true;
                }
            }
            else{
                System.out.println();
                System.out.println("error en la linea de comandos:");
                System.out.println("   sobra el parámetro "+argumentos[i]);
            }
        }
    }
    if (interfaz.getModoAyuda())
        interfaz.ayuda();
    else {

        interfaz.IniciarPrograma();
    }
  }
}
```

## - La clase Banco de pruebas -

```
import  java.io.*;
import  java.util.*;
/**
 * En la clase bateria de pruebas paso al programa una serie de hashis de prueba para ver como
*responde
 */
public class BancoDePruebas
{
   private static long t1, t2;
   private static float dif;

   /**
    * Constructor for objects of class BateriaPruebas
    */
   public  BancoDePruebas()
   {

   }

    public static  void testBateriaDePruebas()
   {
      Calendar ahora1 = Calendar.getInstance();
      t1 = ahora1.getTimeInMillis();
      //Ok
      InterfaceUsuario interfac1 = new InterfaceUsuario();
      interfac1.setNombreFichero("./banco de Pruebas/ejemploHashi7_7(2).txt");
      //interfac1.setModoTraza();
      interfac1.IniciarPrograma();

      //Ok
      InterfaceUsuario interfac2 = new InterfaceUsuario();
      interfac2.setNombreFichero("./banco de Pruebas/ejemploHashi7_7.txt");
      interfac1.setModoTraza();
      interfac2.IniciarPrograma();



      //Ok
      InterfaceUsuario interfac4 = new InterfaceUsuario();
      interfac4.setNombreFichero("./banco de Pruebas/ejemploHashi7_7_n(2).txt");
      //interfac1.setModoTraza();
      interfac4.IniciarPrograma();

      //Ok
      InterfaceUsuario interfac5 = new InterfaceUsuario();
```

```
interfac5.setNombreFichero("./banco de Pruebas/ejemploHashi8_8_n2.txt");
//interfac1.setModoTraza();
interfac5.IniciarPrograma();

//Ok
InterfaceUsuario interfac6 = new InterfaceUsuario();
interfac6.setNombreFichero("./banco de Pruebas/ejemploHashi9_9(2).txt");
//interfac1.setModoTraza();
interfac6.IniciarPrograma();

//Ok
InterfaceUsuario interfac7 = new InterfaceUsuario();
interfac7.setNombreFichero("./banco de Pruebas/ejemploHashi9_9.txt");
//interfac1.setModoTraza();
interfac7.IniciarPrograma();

//Ok
InterfaceUsuario interfac8 = new InterfaceUsuario();
interfac8.setNombreFichero("./banco de Pruebas/ejemploHashi9_9_n.txt");
//interfac1.setModoTraza();
interfac8.IniciarPrograma();


//Ok
InterfaceUsuario interfac10 = new InterfaceUsuario();
interfac10.setNombreFichero("./banco de Pruebas/ejemploHashi11_11.txt");
//interfac1.setModoTraza();
interfac10.IniciarPrograma();

//Ok
InterfaceUsuario interfac11 = new InterfaceUsuario();
interfac11.setNombreFichero("./banco de Pruebas/ejemploHashi13_13(2).txt");
//interfac1.setModoTraza();
interfac11.IniciarPrograma();

//Ok
InterfaceUsuario interfac12 = new InterfaceUsuario();
interfac12.setNombreFichero("./banco de Pruebas/ejemploHashi13_13.txt");
//interfac1.setModoTraza();
interfac12.IniciarPrograma();

//Ok
InterfaceUsuario interfac122 = new InterfaceUsuario();
interfac122.setNombreFichero("./banco de Pruebas/ejemploHashiOficial.txt");
//interfac1.setModoTraza();
interfac122.IniciarPrograma();

//Ok
InterfaceUsuario interfac9 = new InterfaceUsuario();
interfac9.setNombreFichero("./banco de Pruebas/ejemploHashi11_11(2).txt");
//interfac1.setModoTraza();
interfac9.IniciarPrograma();
```

```
//Ok
InterfaceUsuario interfac13 = new InterfaceUsuario();
interfac13.setNombreFichero("./banco de Pruebas/ejemploHashi15_15(2).txt");
//interfac1.setModoTraza();
interfac13.IniciarPrograma();

//Ok
InterfaceUsuario interfac14 = new InterfaceUsuario();
interfac14.setNombreFichero("./banco de Pruebas/ejemploHashi15_15(3).txt");
//interfac1.setModoTraza();
interfac14.IniciarPrograma();

 //Ok
InterfaceUsuario interfac22 = new InterfaceUsuario();
interfac22.setNombreFichero("./banco de Pruebas/ejemploHashi17_17(4).txt");
//interfac1.setModoTraza();
interfac22.IniciarPrograma();

 //Ok
InterfaceUsuario interfac24 = new InterfaceUsuario();
interfac24.setNombreFichero("./banco de Pruebas/ejemploHashi19_19(2).txt");
//interfac1.setModoTraza();
interfac24.IniciarPrograma();

//Ok
InterfaceUsuario interfac33 = new InterfaceUsuario();
interfac33.setNombreFichero("./banco de Pruebas/ejemploHashi23_23(3).txt");
//interfac1.setModoTraza();
interfac33.IniciarPrograma();

//Ok
InterfaceUsuario interfac34 = new InterfaceUsuario();
interfac34.setNombreFichero("./banco de Pruebas/ejemploHashi23_23(4).txt");
//interfac1.setModoTraza();
interfac34.IniciarPrograma();

//Ok
InterfaceUsuario interfac38 = new InterfaceUsuario();
interfac38.setNombreFichero("./banco de Pruebas/ejemploHashi25_25(4).txt");
//interfac1.setModoTraza();
interfac38.IniciarPrograma();



//Ok
InterfaceUsuario interfac15 = new InterfaceUsuario();
interfac15.setNombreFichero("./banco de Pruebas/ejemploHashi15_15(4).txt");
//interfac1.setModoTraza();
interfac15.IniciarPrograma();

//Ok
```

```
InterfaceUsuario interfac16 = new InterfaceUsuario();
interfac16.setNombreFichero("./banco de Pruebas/ejemploHashi15_15.txt");
//interfac1.setModoTraza();
interfac16.IniciarPrograma();

//Ok
InterfaceUsuario interfac17 = new InterfaceUsuario();
interfac17.setNombreFichero("./banco de Pruebas/ejemploHashi17_17(2).txt");
//interfac1.setModoTraza();
interfac17.IniciarPrograma();



//Ok
InterfaceUsuario interfac23 = new InterfaceUsuario();
interfac23.setNombreFichero("./banco de Pruebas/ejemploHashi17_17.txt");
//interfac1.setModoTraza();
interfac23.IniciarPrograma();



//Ok
InterfaceUsuario interfac25 = new InterfaceUsuario();
interfac25.setNombreFichero("./banco de Pruebas/ejemploHashi19_19(3).txt");
//interfac1.setModoTraza();
interfac25.IniciarPrograma();

//Ok
InterfaceUsuario interfac26 = new InterfaceUsuario();
interfac26.setNombreFichero("./banco de Pruebas/ejemploHashi19_19(4).txt");
//interfac1.setModoTraza();
interfac26.IniciarPrograma();

//Ok
InterfaceUsuario interfac27 = new InterfaceUsuario();
interfac27.setNombreFichero("./banco de Pruebas/ejemploHashi19_19(5).txt");
//interfac1.setModoTraza();
interfac27.IniciarPrograma();



//Ok
InterfaceUsuario interfac29 = new InterfaceUsuario();
interfac29.setNombreFichero("./banco de Pruebas/ejemploHashi21_21(2).txt");
//interfac1.setModoTraza();
interfac29.IniciarPrograma();

//Ok
InterfaceUsuario interfac30 = new InterfaceUsuario();
interfac30.setNombreFichero("./banco de Pruebas/ejemploHashi21_21(3).txt");
//interfac1.setModoTraza();
interfac30.IniciarPrograma();
```

```
//Ok
InterfaceUsuario interfac31 = new InterfaceUsuario();
interfac31.setNombreFichero("./banco de Pruebas/ejemploHashi21_21.txt");
//interfac1.setModoTraza();
interfac31.IniciarPrograma();

//Ok
InterfaceUsuario interfac32 = new InterfaceUsuario();
interfac32.setNombreFichero("./banco de Pruebas/ejemploHashi23_23(2).txt");
//interfac1.setModoTraza();
interfac32.IniciarPrograma();



//Ok
InterfaceUsuario interfac35 = new InterfaceUsuario();
interfac35.setNombreFichero("./banco de Pruebas/ejemploHashi23_23.txt");
//interfac1.setModoTraza();
interfac35.IniciarPrograma();



//Ok
InterfaceUsuario interfac37 = new InterfaceUsuario();
interfac37.setNombreFichero("./banco de Pruebas/ejemploHashi25_25(3).txt");
//interfac1.setModoTraza();
interfac37.IniciarPrograma();



//Ok
InterfaceUsuario interfac39 = new InterfaceUsuario();
interfac39.setNombreFichero("./banco de Pruebas/ejemploHashi25_25(5).txt");
//interfac1.setModoTraza();
interfac39.IniciarPrograma();

//Ok
InterfaceUsuario interfac40 = new InterfaceUsuario();
interfac40.setNombreFichero("./banco de Pruebas/ejemploHashi25_25.txt");
//interfac1.setModoTraza();
interfac40.IniciarPrograma();

//Ok
InterfaceUsuario interfac21 = new InterfaceUsuario();
interfac21.setNombreFichero("./banco de Pruebas/ejemploHashi17_17(3).txt");
//interfac1.setModoTraza();
interfac21.IniciarPrograma();



//Ok
```

```
        InterfaceUsuario interfac28 = new InterfaceUsuario();
        interfac28.setNombreFichero("./banco de Pruebas/ejemploHashi19_19.txt");
        //interfac1.setModoTraza();
        interfac28.IniciarPrograma();

        //Ok
        InterfaceUsuario interfac36 = new InterfaceUsuario();
        interfac36.setNombreFichero("./banco de Pruebas/ejemploHashi25_25(2).txt");
        //interfac1.setModoTraza();
        interfac36.IniciarPrograma();

        Calendar ahora2 = Calendar.getInstance();
        t2 = ahora2.getTimeInMillis();

        dif =(float) (t2 - t1) / 1000;
                System.out.println();
                System.out.println("Bateria de pruebas de 41 hashis acabada en " + dif/60 + " minutos "
+ "y " + dif % 60 + " segundos");
                System.out.println("Lo que da una media de  " + dif/40 + " segundos por hashi ");
                System.out.println();

    }
}
```

## - La clase Isla -

```
import java.util.HashSet;
import java.util.ArrayList;

/**
 * Una pequeña clase donde pongo los campos y los metodos utiles para trabajar con
 * las islas.
 */
public class Isla
{
   //codigo para posibilitar la copia (clonacion) de los objetos de esta clase


   private int fila;        //la fila donde esta la isla de la isla
   private int colum;       //la columna donde esta la isla
   private int valor;   //valor numerico de la isla



   //Habra un conjunto de puentes en el que iremos metiendo los puentes que se vayan haciendo
entre islas
   private HashSet<Puente> conjuntoDePuentes = new HashSet<Puente>();

   /**
    * Constructor de la clase
    */
   public Isla(){
   }


   /**
    * Constructor de la clase
    */
    public Isla(int fil, int col, int val)
   {
   fila = fil;
   colum = col;
   valor = val;
   }


   /**
    * Obtiene la fila de una isla dada
    */
  static public int obtenerFila (Isla isla){
```

```
    return isla.fila;
 }

 /**
  * Obtiene la columna de una isla dada
  */
 static public int obtenerColum (Isla isla){
    return isla.colum;
 }

  /**
  * Obtiene el valor de una isla dada
  */
public int obtenerValor (){
    return valor;
 }

 /**
  * Pone el valor de una isla
  */
  void ponerValor(int valorNuevo){

    valor = valorNuevo;
 }

 /**
  * Clonador de Islas
  */
 public Isla clonar(){

    Isla islaClonada = new Isla();

    islaClonada.fila = fila;
    islaClonada.colum = colum;
    islaClonada.valor = valor;

    return islaClonada;
 }

 /**
  * Me da la distancia entre dos islas
  */
 static public int distancia(Isla islaA, Isla islaB){

    int dist = 0;

    if (islaA.fila == islaB.fila)
       dist = Math.abs(islaA.colum - islaB.colum);
    else if (islaA.colum == islaB.colum)
       dist = Math.abs(islaA.fila - islaB.fila);

    return dist;
```

```
}


 /**
 * enLinea devuelve verdadero si dos islas estan en la misma fila o la misma columna
 *  y por tanto son ,en principio, susceptibles de crear un puente entre ellas
 */
static public boolean enLinea (Isla islaA, Isla islaB){

   return ((islaA.fila == islaB.fila) || (islaA.colum == islaB.colum));

}

/**
 * Devuelve verdadero si dos islas tienen las mismas coordenadas
 */
static public boolean iguales(Isla islaA, Isla islaB){

   return( (islaA.fila == islaB.fila) &&
           (islaA.colum == islaB.colum) );

}

/**
 * Devuelve verdadero si dos islas tienen las mismas coordenadas y
 *  los mismos valores
 */
static public boolean identicas(Isla islaA, Isla islaB){

   return( (islaA.fila == islaB.fila) &&
           (islaA.colum == islaB.colum) &&
           (islaA.valor == islaB.valor));

}



/**
 * IslasAlaVista devuelve verdadero si, en principio, se puede tender un puente entre
 *  las dos islas, este mÃ©todo no tiene en cuenta los posibles puentes cruzados que
 *  habrÃ¡n de revisarse con el correspondiente mÃ©todo de la clase Puente
 */
public boolean islasAlaVista(Isla islaA, Isla islaB, ArrayList<Isla> listaDeIslas){

   return ((enLinea(islaA,islaB)) &&
           (!iguales(islaA,islaB)) &&
           (!(Isla.islasCruzadas(islaA,islaB,listaDeIslas))));
  }
```

```
  /**
   * Dada un par de islas mira a ver si hay alguna isla cruzada entre estas dos
   */
  static public boolean islasCruzadas(Isla islaA, Isla islaB, ArrayList<Isla> listaDeIslas)
  {
     boolean cruzadas = false;
     //  bÃºsqueda horizontal
     if(islaA.fila == islaB.fila){
        for(Isla islaMirada : listaDeIslas) {
           if (islaMirada.fila == islaA.fila) {
              if(((islaMirada.colum > islaA.colum) && (islaMirada.colum < islaB.colum)) ||
                 ((islaMirada.colum > islaB.colum) && (islaMirada.colum < islaA.colum))){
                    cruzadas = true;
                    break;
                 }
                 }
              }
           }

     //  bÃºsqueda vertical
     if(islaA.colum == islaB.colum){
        for(Isla islaMirada : listaDeIslas) {
           if (islaMirada.colum == islaA.colum) {
              if(((islaMirada.fila > islaA.fila) && (islaMirada.fila < islaB.fila)) ||
                 ((islaMirada.fila > islaB.fila) && (islaMirada.fila < islaA.fila))){
                    cruzadas = true;
                    break;
                 }
                 }
              }
           }
     return cruzadas;
  }

}
```

## - La clase Cola<Tipo> -

```
import java.util.Vector;

public class Cola <Tipo> {

   //private int          inicio;
   //private int          fin;
   private int    size;
   private Vector<Tipo>    elementos;

   public Cola() {
      super();
      elementos = new Vector<Tipo>();
      //inicio = fin = 0;
      size = 0;
   }

   public boolean colaVacia () {
      //if ( (fin-inicio)==0) {
      if ( size==0) {
         return true;
      }
      return false;
   }

   public void encolar ( Tipo o ) {
      //elementos.add(fin++, o);
      elementos.add(size++, o);
   }
   public void vaciarCola(){
      elementos.clear();
      size = 0;
   }

   public Tipo desencolar () {
       Tipo   retorno;

      try {
         if(colaVacia())
            throw new ErrorColaVacia();
         else {
            //return elementos.get(inicio++);
            retorno = elementos.get(0);
            elementos.remove(0);
```

```
            size--;
            return retorno;
          }
       } catch(ErrorColaVacia error) {
          System.out.println("ERROR: la cola esta vacía");
          return null;
       }
    }

    /*
    public int getFin() {
       return fin;
    }

    public int getInicio() {
       return inicio;
    }
    */
    public int getSize() {
       //return (fin-inicio);
       return (size);
    }
}

@SuppressWarnings("serial")
class ErrorColaVacia extends Exception {
    public ErrorColaVacia() {
       super();
    }
}
```

## - La clase Listas -

```
import java.util.ArrayList;
import java.lang.Comparable;
import java.util.Collections;

/**
 * Una clase que trabaja con listas de islas ordenadas
 */
public class Listas implements Comparable
{

    private  ArrayList<Isla> listaDeIslas;
    private int longitud;


   /**
    * Constructor de la clase
    */
   public Listas()
   {

       listaDeIslas = new ArrayList<Isla>();
       longitud = 0;

   }


   /**
    * Constructor de la clase con parametros
    */
   public Listas(ArrayList<Isla> listaInicial)
   {

       listaDeIslas = listaInicial;
       longitud = listaInicial.size();


   }

   /**
    * Estractor de la longitud
    */
```

```
public int obtenerLongitud(){

    return listaDeIslas.size();
}


/**
 * Estractor de la lista de islas
 */
public ArrayList<Isla> obtenerListaDeIslas(){

    return listaDeIslas;
}



/**
 * Comparador de listas
 */
public int compareTo(Object L) {

    Listas otraLista = (Listas) L;

    if(this.longitud < otraLista.obtenerLongitud())
        return -1;
    else if(this.longitud == otraLista.obtenerLongitud())
        return 0;
    else
        return 1;


}




/**
 * Compara dos ArrayList de islas y devuelve verdadero si ambos tienen las mismas islas
 */
static public boolean igualdadListas (ArrayList<Isla> listaA, ArrayList<Isla> listaB){

    boolean iguales = true;
    int tamanoA = listaA.size();
    int tamanoB = listaB.size();

    if (tamanoA != tamanoB)
        return false;

    Isla [ ] arrayA = new Isla [listaA.size()];
    Isla [ ] arrayB = new Isla [listaB.size()];

    arrayA  = listaA.toArray(arrayA);
```

```
        arrayB  = listaB.toArray(arrayB);

        for (int i = 0; i < tamanoA; i++){

            if(!(Isla.identicas(arrayA[i],arrayB[i]))) {

                iguales = false;
                break;
            }
        }

        return iguales;

    }

    /**
     * Mira si una lista de islas contiene una isla con las coordenadas de la pasada como parametro
     */
    static public boolean contiene(Isla isla, ArrayList<Isla> listaIslas){

        boolean laContiene = false;

        for(Isla islaMirada : listaIslas){

            if(Isla.iguales(islaMirada, isla)){

                laContiene = true;
                break;
            }
        }
        return laContiene;
    }

    /**
     * Borra de una lista de islas la isla que tenga las mismas coordenadas que la pasada como
     * parametro
     */
    static public void borra(Isla isla, ArrayList<Isla> listaIslas){

        for(Isla islaMirada : listaIslas){

            if(Isla.iguales(islaMirada, isla)){

                listaIslas.remove(islaMirada);
                break;
            }
        }
    }
}
```

## - La clase EntSal -

```
/**
 * En esta clase se implementan los metodos para leer un fichero de texto en disco con los datos
 * del tablero e ir introduciendo islas en la lista.
 *
 * Tambien se implementean los metodos de salida de la solucion con los que se imprime el archivo
 * de texto de la solucion.
 *
 */
import java.io.*;
import java.lang.Character;
import java.util.ArrayList;
import java.util.HashSet;

class EntSal{

   private int dimension;

   public EntSal () {
      int dimension = 0;
       }

   public void ponerDimension (int dimension){

      this.dimension = dimension;
   }

   public int obtenerDimension (){

    return dimension;

   }
   /**
    * Dada la ruta en la que se encuentra el archivo del tablero construye la
    *  lista de islas
    */
  public  ArrayList<Isla> creaListaDeIslas(String ruta){

     //File archivo = null;
     FileReader fr = null;
     BufferedReader br = null;
```

```
    ArrayList<Isla> listaDeIslas = new ArrayList<Isla>() ;

    try {
      // Apertura del fichero y creacion de BufferedReader para poder
      // hacer una lectura comoda (disponer del metodo readLine()).
      if (ruta != null)
            br = new BufferedReader(new FileReader(ruta));
      else   {
            br = new BufferedReader(new InputStreamReader(System.in));
            if(!br.ready()){
               System.out.println();
               System.out.println("El fichero introducido no existe, o ha tecleado su nombre
erroneamente");
               System.exit(1);
            }
        }

      char caracterLeido;
      int fila = 0;
      int columna = 0;
      int valor;


      // Lectura del fichero, primero saco la dimension del hashi
      boolean dimensionLeida = false;
      String lineaLeida ;
      int numLeido;

      while(!dimensionLeida){

         lineaLeida = br.readLine();

         if ((lineaLeida.length() > 3) || (lineaLeida.contains("#") ))
            continue;

         numLeido = Integer.parseInt(lineaLeida);

         if( (numLeido > 0) && (numLeido <= 25)){

             ponerDimension(numLeido);
             dimensionLeida = true;
            }

        }
      // pasa por alto la siguiente linea
      lineaLeida = br.readLine();

      caracterLeido =  (char)br.read();
      while(caracterLeido != '+') {

        if (caracterLeido == '\n') {
            ++fila;
```

```
            columna = 0;
        }
        else if (caracterLeido == ' ') {
        //si el cararcer leido es el espacio en blanco no lo cuento
        }
        else if (!(Character.isDigit(caracterLeido))) {
            ++columna; }
        else {
            valor = Character.getNumericValue(caracterLeido);
            // creo una nueva isla con los valores leidos
            Isla nuevaIsla = new Isla(fila,columna,valor);
            // y la meto en la lista de islas
            listaDeIslas.add(nuevaIsla);
            ++columna;
        }
        caracterLeido =  (char)br.read();

      }


   }
   catch(Exception e){
      e.printStackTrace();
   }finally{
      // En el finally cerramos el fichero, para asegurarnos
      // que se cierra tanto si todo va bien como si salta
      // una excepcion.
      try{
         if( null != fr ){

            fr.close();
         }
      }catch (Exception e2){
         e2.printStackTrace();
      }
     }

    return listaDeIslas;
}

/**
 * Imprime un tablero en un archivo de texto  con los los puentes tendidos
 *  para eso se sirve del metodo puentesAarray que pasa primero los puentes
 *  a un array bidimensional
 *
 * Por comodidad creara un array bidimensional con los datos de los puentes
 * que luego imprimira linea por linea con los metodos de impresion
 *
 */
public void imprimePuentes(char [ ][ ] arrayAimprimir){

        for(int i = 0; i <= 3 ; i++)
```

```
            System.out.println();

          //Imprime el array con la informacion de los puentes en el archivo de textolinea por linea
          // esquema a seguir para trabajar con arrays:  for (i = 0; i <= x.length - 1; i++)
          for(int fila = 0; fila <= dimension - 1 ; fila++)
           for(int columna = 0; columna <= dimension - 1 ; columna++){

              if(columna == dimension - 1)
                 System.out.println(arrayAimprimir[fila][columna]);
              else
                 System.out.print(arrayAimprimir[fila][columna] + " ");

           }
    }

    /**
     * Pasa la informacion de los puentes a un array bidimensional que luego sera¡
     * mas facil de imprimir
     *
     * Antes de utilizar propiamente este metodo tengo que hacer la operacion de
     * union entre los puentes Verticales y Horizontales para crear el conjunto de
     * todos los puentes tendidos
     */
    public char [ ] [ ] puentesAarray (HashSet<Puente> puentesTendidos,ArrayList<Isla>
listaDeIslas){

        char [ ][ ] arrayAimprimir = new char[dimension][dimension];
        //lleno al array de espacios en blanco : esquema a seguir para trabajar con arrays:  for (i = 0; i
<= x.length - 1; i++)
        for(int fi = 0; fi <= dimension - 1; fi++)
           for(int co = 0; co <= dimension - 1; co++)
              arrayAimprimir[fi][co] = ' ';

        //Meto las islas en el Array
        int fi;
        int co;
        for(Isla islaMirada : listaDeIslas){

           fi = Isla.obtenerFila(islaMirada);
           co = Isla.obtenerColum(islaMirada);
           String cadenaUnitariaA = Integer.toString (islaMirada.obtenerValor());
           arrayAimprimir[fi][co] = cadenaUnitariaA.charAt(0);
        }

        for(Puente puenteMirado : puentesTendidos) {
           //pone las islas del puente en el array
           int fiA = Isla.obtenerFila(Puente.obtenerIslaA(puenteMirado));
           int coA = Isla.obtenerColum(Puente.obtenerIslaA(puenteMirado));
           int fiB = Isla.obtenerFila(Puente.obtenerIslaB(puenteMirado));
           int coB = Isla.obtenerColum(Puente.obtenerIslaB(puenteMirado));

           //imprimo los arcos del puente en el array
```

```
        int [ ]columnas = Puente.obtenerColumnas(puenteMirado);
        int [ ]filas = Puente.obtenerFilas(puenteMirado);
        Tipo tipoPuente = Puente.obtenerTipo(puenteMirado);
        Orientacion orientacionPuente = Puente.obtenerOrientacion(puenteMirado);

        if(orientacionPuente == Orientacion.horizontal){
           int distColum= Math.abs(coA - coB) - 1 ;
           int columMin = Math.min(coA,coB);
            while(distColum != 0){
              columMin++;
              if(tipoPuente == Tipo.simple)
                 arrayAimprimir[fiA] [columMin] = '-';
              if(tipoPuente == Tipo.doble)
                 arrayAimprimir[fiA] [columMin] = '=';
              distColum--;
           }
        }
        if(orientacionPuente == Orientacion.vertical){
           int distFila = Math.abs(fiA - fiB) - 1 ;
           int filaMin = Math.min(fiA,fiB) ;
           while(distFila != 0){
              filaMin++;
              if(tipoPuente == Tipo.simple)
                 arrayAimprimir[filaMin] [coA ] = '|';
              if(tipoPuente == Tipo.doble)
                 arrayAimprimir[filaMin] [coA] = 'H';
              distFila--;
           }
        }

      }

      return arrayAimprimir;
      }


      /**
       * Imprime en el fichero dado en la ruta la situacion actual del Hashi con los puentes que
       * se hayan tendido y los valores actuales de las islas
       */
      public void imprimirHashi( HashSet<Puente> puentesVerticales, HashSet<Puente>
puentesHorizontales,ArrayList<Isla> listaDeIslas){

          //Hago la union de los puentes verticales y horizontales en un solo conjunto de
          // puentes
          HashSet<Puente> conjuntoTodosLosPuentes = Puente.unirPuentes ( puentesVerticales,
puentesHorizontales);


          char arrayAimprimir [ ] [ ]= puentesAarray (conjuntoTodosLosPuentes,listaDeIslas);
          imprimePuentes(arrayAimprimir);
```

```
    }


}
```

## - La clase Puente -

```
/**
 * En esta clase pongo los metodos que implementan y relacionan el objeto puente
 */
import java.util.ArrayList;
import java.lang.reflect.Array ;
import java.util.HashSet;

enum Tipo {simple, doble};
enum Orientacion {horizontal,vertical};

public class Puente
{
   // instance variables - Un puente enlaza dos islas A y B y puede ser de tipo
   // simple o de tipo doble
   private Isla islaA;
   private Isla islaB;
   Tipo tipoPuente;
   Orientacion orientacion;
   private int [ ]filas ;  //conjunto de filas y columnas  que cruzarÃ¡ el puente
   private int [ ]columnas;

   /**
    * Constructor
    */
   public Puente()
   {
      }

   /**
    * Constructor
    */
   public Puente(Isla islaInicial, Isla islaFinal)
   {
     this.islaA = islaInicial;
     this.islaB = islaFinal;
     this.tipoPuente = Tipo.simple;
     if(Isla.obtenerFila(islaInicial) == Isla.obtenerFila(islaFinal)){
         this.orientacion = Orientacion.horizontal;
```

```
        ponerColumnas();
    }

   else if (Isla.obtenerColum(islaInicial) == Isla.obtenerColum(islaFinal)){
        this.orientacion = Orientacion.vertical;
        ponerFilas();
    }
}

/**
 * Cloneador de puentes
 */
public Puente clonar(){

    Puente puenteClonado = new Puente();

    //El siguiente cÃ³digo no sirve para cambiar el valor de la isla, se modifica tambiÃ©n el de
    // la isla original: puenteClonado.islaA = islaA, si se hace asÃ el
    // clonado y el original compartirian las mismas islas.
    puenteClonado.islaA = islaA.clonar();
    puenteClonado.islaB = islaB.clonar();

    puenteClonado.filas = new int[filas.length];
    puenteClonado.columnas = new int[columnas.length];

    System.arraycopy(filas,0, puenteClonado.filas,0,filas.length);
    System.arraycopy(columnas,0, puenteClonado.columnas,0,columnas.length);

    puenteClonado.tipoPuente = tipoPuente;
    puenteClonado.orientacion = orientacion;


    return puenteClonado;
}


/**
 * Obtiene la islaA de un puente dado
 */
static public Isla obtenerIslaA (Puente puente){
    return puente.islaA;
}

/**
 * Obtiene la islaB de un puente dado
 */
static public Isla obtenerIslaB (Puente puente){
    return puente.islaB;
}

/**
 * Obtiene las filas de un puente dado
```

```
 */
static public int [ ] obtenerFilas (Puente puente){
   return puente.filas;
}

/**
 * Obtiene las columnas de un puente dado
 */
static public int [ ] obtenerColumnas (Puente puente){
   return puente.columnas;
}

/**
 * Obtiene el tipo de puente
 */
static public Tipo obtenerTipo (Puente puente){
   return puente.tipoPuente;
}

/**
 * Obtiene la orientacion del puente
 */
static public Orientacion obtenerOrientacion (Puente puente){
   return puente.orientacion;
}

/**
 * Pone un puente como simple
 */

static public void ponerSimple(Puente puente){

   puente.tipoPuente = Tipo.simple;
}

/**
 * Pone un puente como doble
 */

static public void ponerDoble(Puente puente){

   puente.tipoPuente = Tipo.doble;
}

/**
 * ponerFilas es un metodo auxiliar del constructor que marca las filas que cruza
 *  un puente vertical
 */
private void ponerFilas(){

   columnas = new int [ ] { Isla.obtenerColum(islaA) };
   int distancia = Math.abs (Isla.obtenerFila(islaA) - Isla.obtenerFila(islaB));
```

```
      int valorInicial =  Math.min (Isla.obtenerFila(islaA),Isla.obtenerFila(islaB));
      filas = new int [distancia + 1];
      int i = valorInicial;
      for(int j = 0; j <= distancia; j++){
         filas[j] = i;
         i++;
            }
      }

    /**
    * ponerColumas es un metodo auxiliar del constructor que marca las columnas que cruza
    *  un puente horizontal
    */
   private void ponerColumnas(){

      filas = new int [ ] { Isla.obtenerFila(islaA) };
      int distancia = Math.abs (Isla.obtenerColum(islaA) - Isla.obtenerColum(islaB));
      int valorInicial =  Math.min (Isla.obtenerColum(islaA),Isla.obtenerColum(islaB));
      columnas = new int [distancia + 1];
      int i = valorInicial;
      for(int j = 0; j <= distancia ; j++){
         columnas[j] = i;
         i++;
            }
       }


   /**
    * Rastrea en el conjunto de puentes tendios a ver si hay alguno que conecte la islaInicial
    *  con la islaFinal, si es asi, devuelve el puente que las conecta, en caso contrario devuelve null
    */
   static public Puente mirarPuente(Isla islaInicial, Isla islaFinal, HashSet<Puente>
conjuntoDePuentes){

      Puente puenteExistente = null;
      for(Puente puenteMirado : conjuntoDePuentes) {

         if(((Isla.iguales(puenteMirado.islaA,islaInicial)) &&
(Isla.iguales(puenteMirado.islaB,islaFinal))) ||
            ((Isla.iguales(puenteMirado.islaA, islaFinal))   &&
(Isla.iguales(puenteMirado.islaB,islaInicial)))){
               puenteExistente = puenteMirado;
               break;
            }
            }
      return puenteExistente;

   }

   /**
    * Comprueba si hay un puente doble entre las dos islas en el conjunto de puentes
    */
```

```
   static public boolean puenteDoble (Isla islaInicial, Isla islaFinal, HashSet<Puente>
conjuntoDePuentes){

       boolean doblePuente = false;

       Puente puenteExistente = mirarPuente(islaInicial, islaFinal,conjuntoDePuentes);

       if( (puenteExistente != null) && (puenteExistente.tipoPuente == Tipo.doble))
          doblePuente = true;

       return doblePuente;
   }

   /**
    *Devuelve verdadero si se puede hacer un puente entre las dos islas dadas, mirando
    *a ver si ya hay algun puente previo entre ambas
    */
       static public boolean puenteLibre(Isla islaInicial, Isla islaFinal, HashSet<Puente>
conjuntoDePuentes, ArrayList<Isla> listaIslas) {

          boolean sePuedeTender = true;
          Puente puenteExistente = null;

          // Primero se revisa el conjunto de los puentes a ver si ya hay alguno
          // tendido entre estas dos islas
          puenteExistente = mirarPuente(islaInicial,islaFinal, conjuntoDePuentes);

          //Si lo hay, y es doble, no podemos tender mas puentes entre estas dos islas y se devuelve
falso
          if((puenteExistente != null) && (puenteExistente.tipoPuente == Tipo.doble)) {
             sePuedeTender = false;
             return sePuedeTender;
          }

          if((puenteExistente != null) &&
             (!(Listas.contiene(islaInicial,listaIslas)) || !(Listas.contiene(islaFinal,listaIslas)))){
             sePuedeTender = false;
             return sePuedeTender;
          }

          return sePuedeTender;
       }

   /**
    * Mira a ver si hay algun puente cruzado en el trayecto de la islaA a la islaB
    */
    static public boolean puentesCruzados(Isla islaInicial, Isla islaFinal, HashSet<Puente>
conjPuentesVerticales, HashSet<Puente> conjPuentesHorizontales) {
        //Antes de nada y para que no se produzcan errores en el siguiente algoritmo he de quitar del
        // conjunto de puentes Verticales y Horizontales pasados al mÃ©todo todos aquellos puentes
que
```

```
        // salgan de la isla inicial y de la final

        //Hago una copia local de los subconjuntos que voy a estraer momentaneamente de los
globales
        HashSet<Puente> conjPuentHorInicial = puentesDeLaIsla
(islaInicial,conjPuentesHorizontales);
        HashSet<Puente> conjPuentVerInicial   = puentesDeLaIsla
(islaInicial,conjPuentesVerticales);
        HashSet<Puente> conjPuentHorFinal = puentesDeLaIsla (islaFinal,conjPuentesHorizontales);
        HashSet<Puente> conjPuentVerFinal   = puentesDeLaIsla (islaFinal,conjPuentesVerticales);

        //Hago la diferencia de conjuntos entre los totales y los subconjuntos que tienen puentes
        // que parten de las islas consideradas
        conjPuentesHorizontales.removeAll(puentesDeLaIsla (islaInicial,conjPuentesHorizontales));
        conjPuentesVerticales.removeAll(puentesDeLaIsla (islaInicial,conjPuentesVerticales));
        conjPuentesHorizontales.removeAll(puentesDeLaIsla (islaFinal,conjPuentesHorizontales));
        conjPuentesVerticales.removeAll(puentesDeLaIsla (islaFinal,conjPuentesVerticales));

        boolean cruzados = false;
        // tiendo un puente provisional entre las dos islas
        Puente puenteProvisional = new Puente(islaInicial,islaFinal);
        // miro en el conjunto de los puentes ya tendidos a ver si alguno interfiere con las filas o
columnas
        // del puente provisional
        if(puenteProvisional.orientacion == Orientacion.horizontal){
            for(Puente puenteMirado : conjPuentesVerticales) {
                //miro si la fila del puente provisional estÃ¡ dentro de las filas del puente mirado
                for(int i = 0; i < Array.getLength(puenteMirado.filas); i++) {
                    if(puenteProvisional.filas[0] == puenteMirado.filas[i]){
                      //miro si alguna de las columnas del puente provisional coincide con las del puente
mirado
                      for(int j = 0; j < Array.getLength(puenteProvisional.columnas); j++){
                        if(puenteProvisional.columnas[j] == puenteMirado.columnas[0]){
                          cruzados = true;
                          break;
                        }

                        }
                      }
                  }
              }
          }
        if(puenteProvisional.orientacion == Orientacion.vertical){
            for(Puente puenteMirado : conjPuentesHorizontales) {
                //miro a ver si alguna de las columnas del puente mirado coincide con la columna del
puente provisonal
                for(int i = 0; i < Array.getLength(puenteMirado.columnas); i++) {
                    if(puenteProvisional.columnas[0] == puenteMirado.columnas[i]){
                      //miro si alguna de las filas del puente provisional coincide con la del puente mirado
                      for(int j = 0; j < Array.getLength(puenteProvisional.filas); j++){
                        if(puenteProvisional.filas[j] == puenteMirado.filas[0]){
                          cruzados = true;
```

```
                        break;
                    }
                    }
                }
            }
        }
    }

    // Deshago los cambios hechos en los conjuntos de puentes uniendo los subconjuntos con los
globales
    conjPuentesHorizontales.addAll(conjPuentHorInicial);
    conjPuentesVerticales.addAll(conjPuentVerInicial);
    conjPuentesHorizontales.addAll(conjPuentHorFinal);
    conjPuentesVerticales.addAll(conjPuentVerFinal);
    return cruzados;
    }


  /**
   * Con este metodo actualizamos el conjunto de los puentes metiendo un nuevo puente al que
   * introduce en el conjuntoDePuentes o que ha encontrado un puente simple en el
   * conjunto de puentes , en cuyo caso lo pone doble.
   *
   * AtenciÃ³n! para usar este mÃ©todo primero nos tenemos que asegurar con el mÃ©todo
puenteLibre
   *  de que se puede tender un puente entre las islas consideradas
   *
   */
  static public void actualizarPuentes(Isla islaInicial, Isla islaFinal, HashSet<Puente>
conjuntoDePuentes)
  {

    Puente puenteExistente = mirarPuente(islaInicial,islaFinal, conjuntoDePuentes);

    //Si no hay puente tendido entre estas dos islas, lo creamos y lo
    // metemos en el conjunto de puentes
    if(puenteExistente == null){
       Puente puenteTendido;
       puenteTendido = new Puente (islaInicial,  islaFinal);
       conjuntoDePuentes.add(puenteTendido);
       }
    //Si lo hay, y es simple, lo ponemos como doble
    else if(puenteExistente.tipoPuente == Tipo.simple){
          puenteExistente.tipoPuente = Tipo.doble;
       }

  }



  /**
   * Me da el conjunto de puentes que salen de la isla considerada
```

```
    */
   static public HashSet<Puente> puentesDeLaIsla (Isla isla, HashSet<Puente> conjuntoDePuentes)
{

      HashSet<Puente> puentesDesdeLaIsla = new HashSet<Puente>();

      for(Puente puenteMirado : conjuntoDePuentes) {
         if((Isla.iguales (puenteMirado.islaA , isla)) || (Isla.iguales (puenteMirado.islaB , isla)))
               puentesDesdeLaIsla.add(puenteMirado);
         }
      return puentesDesdeLaIsla;

   }

   /**
    * Me da la lista de las islas que componen un conjunto de puentes
    */
   static public ArrayList<Isla> islasDeLosPuentes (HashSet<Puente> conjuntoDePuentes){

      ArrayList<Isla> listaDeLasIslas = new ArrayList<Isla>();

      for(Puente puenteMirado : conjuntoDePuentes) {
         if(!(listaDeLasIslas.contains(puenteMirado.islaA)))
               listaDeLasIslas.add(puenteMirado.islaA);

         if(!(listaDeLasIslas.contains(puenteMirado.islaB)))
               listaDeLasIslas.add(puenteMirado.islaB);
         }
      return listaDeLasIslas;

   }

   /**
    * Me da una cuenta de  las vecinas que tienen puentes dirigidos a la isla considerada y que
pueden tender
    * algun puente mas
    */
  static public int vecinasConPuentesAlaIsla (Isla isla,
                                    HashSet<Puente> ConjuntoVerticales,
                                    HashSet<Puente> ConjuntoHorizontales,
                                    ArrayList<Isla> listaIslas){

      int cuenta =  0;
      HashSet<Puente> TodosLosPuentes = unirPuentes (ConjuntoVerticales,
ConjuntoHorizontales);

      ArrayList<Isla> vecinasConPuentesParalaIsla = new ArrayList<Isla>();
      vecinasConPuentesParalaIsla = islasDeLosPuentes (puentesDeLaIsla (isla,TodosLosPuentes));

      ArrayList<Isla> consideradas = new ArrayList<Isla>();

      for(Isla vecinaMirada : vecinasConPuentesParalaIsla){
```

```
        if( !(puenteLibre(isla,vecinaMirada,TodosLosPuentes,listaIslas)) ||
            (Isla.iguales(vecinaMirada, isla)) ||
            (consideradas.contains(vecinaMirada)))
            continue;

        cuenta++;
        consideradas.add(vecinaMirada);
      }

    return cuenta;

  }

  /**
   * Da como resultado la union de dos conjuntos de puentes
   */
  static public  HashSet<Puente> unirPuentes (HashSet<Puente> puentesVerticales,
HashSet<Puente> puentesHorizontales){

    //Hago la uniÃ³n de los puentes verticales y horizontales en un solo conjunto de
        // puentes
        HashSet<Puente> conjuntoTodosLosPuentes =new HashSet<Puente>();
        conjuntoTodosLosPuentes.addAll(puentesVerticales);
        conjuntoTodosLosPuentes.addAll(puentesHorizontales);

        return conjuntoTodosLosPuentes;
  }



  /**
   * Devuelve verdadero si dos puentes son iguales
   */
  static public boolean iguales(Puente puenteA, Puente puenteB){

     return ( (puenteA.islaA == puenteB.islaA) &&
              (puenteA.islaB == puenteB.islaB) &&
              (puenteA.tipoPuente == puenteB.tipoPuente) );

  }

}
```

## - La clase Conjunto_Puentes -

```
import java.util.HashSet;
/**
 * Esta clase implementa un conjunto del tipo HashSet<Puente> con los puentes
 */
public class Conjuto_Puentes
{
    // instance variables - replace the example below with your own
    public HashSet<Puente> conjuntoPuentes;

    /**
     * Constructor for objects of class Conjuto_Puentes
     */
    public Conjuto_Puentes()
    {
        // initialise instance variables
        conjuntoPuentes = new HashSet<Puente>();
    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    public void meterPuente(Puente puente)
    {
        // put your code here
        conjuntoPuentes.add(puente);
    }
}
```