Sistemas inteligentes

Memoria practica extraordinaria

Contenido

Clases del programa

Acción

Memoria practica extraordinaria

La clase accion contiene la posicion X e Y del tractor junto con el coste asociado a realizar esa acción y contiene una lista de distribuciones que muestran como esta el terreno en sus posiciones contiguas.

Variables:

- PosX: coordenada X del tractor.
- PosY: coordenada Y del tractor.
- Coste: el coste asociado a esa acción.
- ListaDistribución: como están las posiciones contiguas de la posición del tractor.

Métodos:

Los métodos que contiene son los sets y los gets de cada variable.

Ejemplo:

```
[posX=0, posY=1, coste=1, listaDistribucion=[[posX=1, posY=2, arena=0], [posX=0, posY=1, arena=0]]]
```

```
--Terreno con posicion del tractor--
|  3   |1|   0  |
|  0    4    0  |
|  0    1    0  |
```

El tractor se moverá a la posición (0, 1) con un coste de 1 y en las posiciones (1, 2) y (0, 1) se añadirán 0 de tierra a la cantidad almacenada.

Distribución

Esta clase indica cuanta arena tiene la posición X e Y.

Variables:

- PosX: coordenada X.
- PosY: coordenada Y.
- Arena: cantidad de tierra que tiene esa posición.

Métodos:

Los métodos que contiene son los sets y los gets de cada variable.

Ejemplo:

```
[posX=1, posY=2, arena=0]
```

La posición (1, 2) tiene 0 de tierra.

Estado

En esta clase se define un estado, define como está el terreno en un momento determinado, como esta la tierra repartida en todas las casillas y las coordenadas de donde se encuentra el tractor.

Variables:

- X: número de filas del terreno.
- Y: número de columnas del terreno.
- V: tierra total en el terreno.

- K: número mínimo de tierra en cada posición del terreno.
- Max: número máximo de tierra en cada posición del terreno.
- Terreno [] []: matriz de posiciones que contiene información de todas las celdas del terreno.
- Datos: Conjunto de datos sobre el terreno.

Métodos:

Los métodos que contiene son los sets y los gets de cada variable junto con otros auxiliares que son:

```
public boolean isRepartido() {
    /**
     * Comprueba si la totalidad de la tierra es "k"
     */
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno[i].length; j++) {
            if (Terreno[i][j].getTierra() != k) {
                return false;
            }
        }
    }
    return true;
}
```

Comprueba en el terreno si todas sus posiciones tienen K, es decir, el mínimo de tierra.

Retorno:

- False: si hay alguna posición que no tenga el mínimo.
- True: si todas las posiciones tienen el mínimo.

```
public String convertirCadena() {
    /**
     * Convertir a String
     */
    String cadena = "";
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno.length; j++) {
            cadena = cadena + (char) Terreno[i][j].getTierra();
        }
    }
    cadena = cadena + PosX;
    cadena = cadena + PosY;
    return cadena;
}
```

Crea un String uniendo la tierra de cada posición del terreno y la posición X e Y del tractor y lo retorna.

```
public int CalculoHeuristica() {
    /**
     * calculo de heuristica
     */
    int heuristica = 0;
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno[i].length; j++) {
            if (Terreno[i][j].getTierra() != k) {
                heuristica = heuristica + 1;
            }
        }
    }
    return heuristica;
}
```

Calcula la heurística como la suma de todas las casillas que no tiene cantidad mínima, es decir, la K.

Retorno:

- Heurística: el sumatorio de las casillas que no tienen K.

```
public Posicion[][] copia() {
    Posicion[][] copiaP = new Posicion[this.x][this.y];
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno[i].length; j++) {
            copiaP[i][j] = new Posicion(Terreno[i][j].getPosX(), Terreno[i][j].getPosY(), Terreno[i][j].getTierra(),
                    Terreno[i][j].getListaDistribucion());
        }
    }
    return copiaP;
}

public Estado clone() {
    Estado clon = new Estado(this.getDatos(), copia(), this.getPosX(), this.getPosY());
    clon.setK(this.k);
    clon.setMax(max);
    clon.setV(this.V);
    clon.setX(this.x);
    clon.setY(this.y);
    return clon;
}
```

Métodos usados para la clonación de un estado.

Retorno:

- Método: 1º una matriz de terreno clonada.
- Método: 2º un objeto Estado clonado.

```
public int CalcularTierraSobrante() {
    /**
     * retorna la tierra sobrante en una posicion determinada
     */
    int terreno = Terreno[getPosX()][getPosY()].getTierra() - getK();
    if (terreno <= 0) {
        return 0;
    } else {
        return terreno;
    }
}
```

Para una posición del tractor obtiene la tierra que se puede mover, es decir, la tierra que hay en la posición X e Y menos el mínimo de tierra.

Retorno:

- Si la tierra menos K es negativa se retorna 0.
- Si la tierra menos K es positiva se retorna la tierra.

```
public void realizarAccion(Accion a) {
    int fila;
    int columna;
    int arena;
    int arenaRestante = 0;
    for (int i = 0; i < a.getListaDistribucion().size();i++) {
        fila = a.getListaDistribucion().get(i).getPosX();
        columna = a.getListaDistribucion().get(i).getPosY();
        arena = a.getListaDistribucion().get(i).getArena();
        arenaRestante = arenaRestante + arena;
        getTerreno()[fila][columna].setTierra(getTerreno()[fila][columna].getTierra() + arena);
    }
    getTerreno()[getPosX()][getPosY()].setTierra(getTerreno()[getPosX()][getPosY()].getTierra() - arenaRestante);
    setPosX(a.getPosX());
    setPosY(a.getPosY());
}
```

Dada una acción genera un nuevo estado, modificando la tierra del terreno y la posición del tractor.

Ejemplo:

```
Estado [x=3, y=3, V=9, k=1, max=4, PosX=0, PosY=2, datos=[3, 3, 4, 1, 9]]
--Terreno con posicion del tractor--
|  3    1   |0| |
|  0    4    0  |
|  0    1    0  |
```

Ficheros

Esta clase se encarga de leer, escribir y borrar los datos contenidos en los ficheros necesarios para ejecutar el programa.

Métodos:

```
public void BorrarFicheroSolucion() {
    try {
        File file = new File("SolucionesTerreno.txt");
        BufferedWriter bw = new BufferedWriter(new FileWriter(file.getAbsolutePath()));
        bw.write("");
        bw.close();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Este método borra lo contenido en el fichero SolucionesTerreno.txt.

```
public ArrayList<Integer> LecturaFicheroTerrenoPrefijado() {
    ArrayList<Integer> contenido = new ArrayList<Integer>();
    Scanner datos;
    try {
        File f = new File("DatosTerrenoPrefijado.txt");
        datos = new Scanner(new FileReader(f.getAbsolutePath()));
        while (datos.hasNext()) {
            int X = datos.nextInt();
            System.out.println("    Número de filas del terreno: " + X);
            contenido.add(X);
            int Y = datos.nextInt();
            System.out.println("    Número de columnas del terreno: " + Y);
            contenido.add(Y);
            int maximo = datos.nextInt();
            System.out.println("    Valor de tierra máximo en el terreno: " + maximo);
            contenido.add(maximo);
            int k = datos.nextInt();
            System.out.println("    Valor de tierra mínimo en el terreno: " + k);
            contenido.add(k);
            int v = X * Y * k;
            contenido.add(v);
            System.out.println("    Tierra total del terreno: " + v);
        }
    } catch (Exception e) {
        System.out.println("Error de lectura ");
        System.out.println(e.toString());
    }
    return contenido;
}
```

Este método lee del fichero DatosTerrenoPrefijado.txt los muestra por pantalla y los almacena en un arrayList.

Retorno:

- ArrayList con todos los datos leídos del fichero.

```
public void EscrituraSolucionFichero(String Terreno[][], int filas, int columnas, int k, int maximo, int posX,
        int posY) {
    try {

        File file = new File("SolucionesTerreno.txt");
        FileWriter fos = new FileWriter(file.getAbsoluteFile(), true);
        BufferedWriter bw = new BufferedWriter(fos);
        bw.write("" + posX);
        bw.write("" + posY);
        bw.write("" + k);
        bw.write("" + maximo);
        bw.write("" + filas);
        bw.write("" + columnas);
        bw.newLine();
        for (int i = 0; i < Terreno.length; i++) {
            for (int j = 0; j < Terreno[i].length; j++) {
                bw.write(Terreno[i][j]);
            }
            bw.newLine();
        }
        bw.newLine();
        bw.close();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Este método escribe de manera continua en un fichero llamado SolucionesTerreno.txt

Frontera

En esta clase se crea una cola de nodos que serán la solución al problema

Variables:

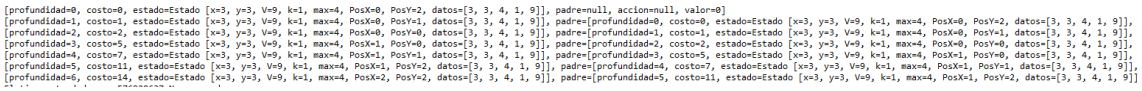
- FronteraCola: cola de objetos nodos.

Métodos:

Los métodos que contiene son los sets y los gets de cada variable, junto con los añadir un objeto a la cola, eliminar el primer objeto de la cola, retornar el tamaño de la cola y comprobar si la cola está vacía.

```
public class Frontera {

    private Queue<Nodo> fronteraCola;

    public Frontera() {
        fronteraCola=new LinkedList<Nodo>();
    }

    public Queue<Nodo> getFronteraCola() {
        return fronteraCola;
    }

    public void setFronteraCola(LinkedList<Nodo> fronteraCola) {
        this.fronteraCola = fronteraCola;
    }

    public void insertarNodo(Nodo n) {
        this.fronteraCola.add(n);
    }
    public Nodo eliminarNodo() {
        Nodo n = this.fronteraCola.poll();
        return n;
    }

    public int getFronteraSize() {
        return fronteraCola.size();
    }

    public boolean esVacia() {
        if (this.fronteraCola.size() > 0)
            return false;
        else
            return true;
    }

    public void limpiar() {
        fronteraCola.clear();
    }
```

Ejemplo:



Conjunto de nodos que son la solución del problema, mostrando la profundidad, costo, estado, el padre, la acción realizada y la lista de distribución asociada a la acción

Nodo

Define un nodo del árbol de búsqueda

Variables:

- Profundidad: profundidad del árbol de búsqueda
- Costo: costo del nodo
- Estado: estado actual
- Padre: estado padre del estado actual
- Acción: acción que se ha realizado para ese estado
- Valor: valor de la heurística

Métodos

Esta clase contiene los sets y los gets de las variables.

Ejemplo:

```
[profundidad=0, costo=0, estado=Estado [x=3, y=3, V=9, k=1, max=4, PosX=0, PosY=2, datos=[3, 3, 4, 1, 9]], padre=null, accion=null, valor=0]
```

Posición

Una posición está definida por la posición X e Y del tractor y una lista de distribución de las posiciones del terreno y un bolean que indica si se ha visitado.

Variables:

- PosX: posición X del tractor.
- PosY: posición Y del tractor.
- ListaDistribución: como están las posiciones contiguas de la posición del tractor.
- Visitado: indica si se ha visitado esa posición

Métodos:

Esta clase contiene los sets y los gets de las variables.

Problema

Un problema esta definido por un estado y un espacio de estados.

Variables:

- EstadoInicial: objeto estado
- EspacioEstados: objeto espacioDeEstados.

Métodos:

Esta clase contiene los sets y los gets de las variables.

Sucesor

Clase que define los atributos de un sucesor, la acción que lo genera, el nuevo estado y el coste de aplicar la acción.

Variables:

- Acción: objeto acción, la acción que se va a realizar.
- Estado: objeto estado, el nuevo estado.
- Coste: conste de realizar la acción.

Métodos:

Esta clase contiene los sets y los gets de las variables.

Serializar

Esta clase encripta por MD5, clase copiada de internet.

Autor de la clase: http://lineasycodigos.blogspot.com.es/2013/01/encriptacion-en-md5-usando-java.html

Generar Terreno

En esta clase se genera el terreno del problema, el terreno es una matriz bidimensional de objetos posiciones que tienen cada posición una cantidad de tierra y un conjunto de posiciones que son las coordenadas a las que tiene acceso directo.

Variables:

- Filas: número de filas del terreno.
- Columnas: número de columnas del terreno.
- Máximo: valor máximo de tierra que debe tener cada posición del terreno.
- K: valor mino de tierra que debe tener cada posición del terreno.
- V: cantidad de tierra total que tiene el terreno.

Métodos:

```
public ArrayList<Distribucion> GenerarTerrenosVecinos(int X, int Y, int filas, int columnas) {
    /**
     * Obtiene cuales son las posiciones contiguas
     */
    ArrayList<Distribucion> vecinos = new ArrayList<Distribucion>();
    if (X < filas - 1) {
        Distribucion Inferior = new Distribucion(X + 1, Y, 0);
        vecinos.add(vecinos.size(), Inferior);
    }
    if (Y > 0) {
        Distribucion Izquierda = new Distribucion(X, Y - 1, 0);
        vecinos.add(vecinos.size(), Izquierda);
    }
    if (X > 0) {
        Distribucion Arriba = new Distribucion(X - 1, Y, 0);
        vecinos.add(vecinos.size(), Arriba);
    }
    if (Y < columnas - 1) {
        Distribucion Derecha = new Distribucion(X, Y + 1, 0);
        vecinos.add(vecinos.size(), Derecha);
    }
    return vecinos;
}
```

Este método dado una posición obtiene cual serían las posiciones adyacentes a las coordenadas introducidas

Retorno

- Retorna un ArrayList donde se almacenan los vecinos.

Ejemplo generar vecinos:

```
--Terreno con posicion del tractor--
|  3   |1|   0  |
|  0    4    0  |
|  0    1    0  |
```

Dado esa posición la (0, 1) el método retornaría (0, 0) – (1, 1) – (0, 2)

```
--Terreno con posicion del tractor--
|  3   |1|   0  |
|  0    4    0  |
|  0    1    0  |
```

Las posiciones a las que se tiene acceso desde la posición indicada

```
public void GenerarVecinos(int filas, int columnas, Posicion[][] Terreno) {
    /**
     * Reparte la tierra en cada zona
     */
    int aux = this.v;
    int cantidad_metida = 0;
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno[i].length; j++) {
            ArrayList<Distribucion> vecinos = GenerarTerrenosVecinos(i, j, filas, columnas);
            cantidad_metida = (int) (Math.random() * this.maximo + 1);
            if (aux - cantidad_metida > 0) {
                aux -= cantidad_metida;
                Posicion n = new Posicion(i, j, cantidad_metida, vecinos);
                Terreno[i][j] = n;
            } else if (aux - cantidad_metida < 0) {
                Posicion n = new Posicion(i, j, aux, vecinos);
                Terreno[i][j] = n;
                aux = 0;
            } else {
                Posicion n = new Posicion(i, j, 0, vecinos);
                Terreno[i][j] = n;
            }
        }
    }
}
```

Este método genera la cantidad de tierra que debe tener cada posición del terreno, el método recorre la matriz, para cada posición obtiene sus vecinos y genera un valor aleatorio de tierra que intenta meter en el terreno,

Condicionales:

- Con el primer condicional comprueba que la tierra máxima (aux) menos la tierra generada sea superior a 0, si es así la introduce en el terreno y la resta a la cantidad total de tierra (aux).
- Con el segundo condicional comprobamos que si la tierra total (aux) menos la tierra generada es negativa, meteremos toda la tierra total restante (aux) y modificaremos el valor de la tierra total restante (aux) a 0 ya que se ha repartido toda la tierra.
- Con el tercer condicional comprobamos para el resto posible de casos, es decir, que el resto sea 0, en caso de ser 0 el resto, la tierra que se almacena es 0.

Ejemplo generar tierra:

```
--Terreno con posicion del tractor--
|  3    1   |0| |
|  0    4    0  |
|  0    1    0  |
```

La tierra total en este caso es 9, (3+1+4+1=9), en la primera posición (0, 0) se obtuvo un 3, la tierra restante para las siguientes posiciones seria de 7 unidades, en la posición (0, 1) se obtuvo un 1, por tanto la tierra restante es de 5 unidades, en las posiciones (0, 2),(1, 0),(2, 2) y (0, 2) se obtuvo un 0, en la posición (1, 1) se obtuvo un 4, por tanto la tierra total restante es de 1 unidad, en la posición (2, 1) se obtuvo un 1 y la tierra restante paso a ser 0, por eso en la posición (2, 2) es 0 ya que no había mas tierra que repartir.

Espacio de estados.

En esta clase se generar todos los estados posibles para un estado determinado.

Métodos:

```
public ArrayList<Accion> obtenerAcciones( Nodo nodo) {
    /**
     * Obtiene todas las acciones para un estado
     */
    GenerarEstados estados = new GenerarEstados();
    ArrayList<Distribucion> posicionesValidas = nodo.getEstado().getMovimientos();
    ArrayList<Accion> listaAcciones = new ArrayList<Accion>();
    ArrayList<Accion> listaTodasAcciones = new ArrayList<Accion>();
    ArrayList<Distribucion> listaDistribuciones = new ArrayList<Distribucion>();
    Estado estado=nodo.getEstado().clone();
    estados.BackPosiblesAcciones(0, posicionesValidas, listaDistribuciones, listaAcciones,estado);
    estados.accionesPosibles(listaAcciones, posicionesValidas, listaTodasAcciones);
    return listaTodasAcciones;
}
```

Este método retorna una lista de todas las posibles acciones para un estado determinado.

Ejemplo acciones:

```
Acciones del: Estado [x=3, y=3, V=9, k=1, max=4, PosX=2, PosY=2, datos=[3, 3, 4, 1, 9]]
[posX=2, posY=1, coste=1, listaDistribucion=[[posX=2, posY=1, arena=0], [posX=1, posY=2, arena=0]]]
[posX=1, posY=2, coste=1, listaDistribucion=[[posX=2, posY=1, arena=0], [posX=1, posY=2, arena=0]]]
```

Para la posición (2, 2) esquina inferior derecha, puede acceder a 2 posiciones (2, 1) y (1, 2), en la posición (2, 2) hay 0 de tierra por tanto las combinaciones posibles serán enviar 0 de tierra a sus 2 posiciones contiguas

```
public ArrayList<Sucesor> obternerSucesores(ArrayList<Accion> listaTodasAcciones, Nodo nodo) {
    /**
     * Aplica las acciones
     */
    ArrayList<Sucesor> listaSucesores = new ArrayList<Sucesor>();
    for (int i = 0; i < listaTodasAcciones.size(); i++) {
        Estado estado = nodo.getEstado().clone();
        estado.realizarAccion(listaTodasAcciones.get(i));
        Sucesor sucee = new Sucesor(listaTodasAcciones.get(i), estado, listaTodasAcciones.get(i).getCoste() + nodo.getCosto());
        listaSucesores.add(sucee);
    }
    return listaSucesores;
}
```

En este método para todas las acciones generadas, las realiza y crea un objeto sucesor que será una posible solución al problema

Ejemplo sucesor:

```
[posX=1, posY=0, coste=3, listaDistribucion=[[posX=1, posY=0, arena=2], [posX=0, posY=1, arena=0]]]
```

Generar Estados

Esta clase es la encargada de generar los estados.

Métodos

```
public void CostoAccion(Accion accion) {
    /**
     * coste asociado a una accion
     */
    int coste = 1;
    for (int i = 0; i < accion.getNumListDistrib(); i++) {
        coste = coste + accion.getDist(i).getArena();
    }
    accion.setCoste(coste);
}
```

En este método se asigna un coste a la acción, en este caso es la tierra que se mueve.

```
public boolean ComprobarPosibleAccion(int valor, ArrayList<Distribucion> listaDistribucionTerreno) {
    /**
     * Solo si se reparte toda la tierra es una accion posible
     */
    int sumatorio = 0;
    boolean salida = false;
    for (int i = 0; i < listaDistribucionTerreno.size(); i++) {
        sumatorio = sumatorio + listaDistribucionTerreno.get(i).getArena();
    }
    if (sumatorio == valor) {
        salida = true;
    }
    return salida;
}
```

En este método se comprueba si es posible realizar ese movimiento, es decir, que la tierra que se quiere mover es la tierra que se tiene que mover exactamente.

```
public boolean ProbarLimiteDeTierra(int cantidadTierra, Distribucion posicion,Estado estado) {
    /**
     * Comprueba si es posible es reparticion de tierra sin superar el maximo
     */
    if ((cantidadTierra + posicion.getArena()) > estado.getMax()) {
        return false;
    } else {
        return true;
    }
}
```

En este método se comprueba que la arena que se intenta mover no sea superior al limite máximo de tierra por posición.

```
public void BackPosiblesAcciones(int etapa, ArrayList<Distribucion> posValidas,
        ArrayList<Distribucion> distribuciones, ArrayList<Accion> acciones,Estado estado) {
    /**
     * Genera las acciones posibles asociadoas a un estado
     */
    if (etapa == posValidas.size()) {
        if (ComprobarPosibleAccion(estado.CalcularTierraSobrante(), distribuciones)) {
            ArrayList<Distribucion> nuevaDistribucion = new ArrayList<Distribucion>(distribuciones);
            Accion nuevaAccion = new Accion(posValidas.get(etapa - 1).getPosX(),
                    posValidas.get(etapa - 1).getPosY(), 1, nuevaDistribucion);
            CostoAccion(nuevaAccion);
            acciones.add(nuevaAccion);
        }
    } else {
        for (int i = 0; i <= estado.CalcularTierraSobrante(); i++) {
            if (ProbarLimiteDeTierra(i, posValidas.get(etapa),estado)) {
                Distribucion distribucion = new Distribucion(posValidas.get(etapa).getPosX(),
                        posValidas.get(etapa).getPosY(), i);
                distribuciones.add(etapa, distribucion);
                BackPosiblesAcciones(etapa + 1, posValidas, distribuciones, acciones,estado);
                distribuciones.remove(etapa);
            }
        }
    }
}
```

Este método genera todas las posibles combinaciones de tierra en varias posiciones.

Ejemplo:

```
[posX=1, posY=0, arena=0], [posX=0, posY=1, arena=2]]
[posX=1, posY=0, arena=1], [posX=0, posY=1, arena=1]]
[posX=1, posY=0, arena=2], [posX=0, posY=1, arena=0]]
```

```
public void accionesPosibles(ArrayList<Accion> listaAcciones, ArrayList<Distribucion> posicionesValidas,
        ArrayList<Accion> listaTodasAcciones) {
    /**
     * genera las acciones posibles, para cada posible posicion copia los generado
     * en el backtracking
     */
    int fila;
    int columna;
    int sizeInicial = listaAcciones.size();
    for (int i = 0; i < posicionesValidas.size(); i++) {
        fila = posicionesValidas.get(i).getPosX();
        columna = posicionesValidas.get(i).getPosY();
        for (int h = 0; h < sizeInicial; h++) {
            Accion a = new Accion(listaAcciones.get(h).getPosX(), listaAcciones.get(h).getPosY(),
                    listaAcciones.get(h).getCoste(), listaAcciones.get(h).getListaDistribucion());
            a.setPosX(fila);
            a.setPosY(columna);
            listaTodasAcciones.add(a);
        }
    }
}
```

Este método “copia” las combinaciones generas por el bactracking para las posiciones adyacentes a la posición actual del tractor.

Ejemplo:

```
[posX=1, posY=0, coste=3, listaDistribucion=[[posX=1, posY=0, arena=0], [posX=0, posY=1, arena=2]]]
[posX=1, posY=0, coste=3, listaDistribucion=[[posX=1, posY=0, arena=1], [posX=0, posY=1, arena=1]]]
[posX=1, posY=0, coste=3, listaDistribucion=[[posX=1, posY=0, arena=2], [posX=0, posY=1, arena=0]]]
[posX=0, posY=1, coste=3, listaDistribucion=[[posX=1, posY=0, arena=0], [posX=0, posY=1, arena=2]]]
[posX=0, posY=1, coste=3, listaDistribucion=[[posX=1, posY=0, arena=1], [posX=0, posY=1, arena=1]]]
[posX=0, posY=1, coste=3, listaDistribucion=[[posX=1, posY=0, arena=2], [posX=0, posY=1, arena=0]]]
```

```
public String[][] CopiarTerreno(Posicion Terreno[][], String TerrenoString[][], ArrayList<Integer> datos) {
    /**
     * Copia los datos de los objetos del terreno a un String
     */
    TerrenoString = new String[datos.get(0)][datos.get(1)];
    for (int i = 0; i < Terreno.length; i++) {
        for (int j = 0; j < Terreno[i].length; j++) {
            TerrenoString[i][j] = "" + Terreno[i][j].getTierra();
        }
    }
    return TerrenoString;
}

public String[][] CopiarTerreno2(Estado e) {
    /**
     * Otro metodo de copiar el terreno
     */
    String[][] terrenoString = new String[e.getX()][e.getY()];
    for (int i = 0; i < e.getTerreno().length; i++) {
        for (int j = 0; j < e.getTerreno()[i].length; j++) {
            terrenoString[i][j] = "" + e.getTerreno()[i][j].getTierra();
        }
    }
    return terrenoString;
}
```

Estos métodos crean una matriz String que contiene la cantidad de tierra de cada posición.

```
public void MostrarTerrenoTractor(int X, int Y, String[][] TerrenoString) {
    /**
     * Mostrar el terreno dada la posición del tractor
     */
    System.out.println("--Terreno con posicion del tractor--");
    for (int i = 0; i < TerrenoString.length; i++) {
        System.out.print("|");
        for (int j = 0; j < TerrenoString[i].length; j++) {
            if (i == X && j == Y) {
                if (TerrenoString[i][j].length() == 1) {
                    System.out.print(" |" + TerrenoString[i][j] + "| ");
                } else {
                    System.out.print("|" + TerrenoString[i][j] + " |");
                }
            } else {
                if (TerrenoString[i][j].length() == 1) {
                    System.out.print("  " + TerrenoString[i][j] + "  ");
                } else {
                    System.out.print(" " + TerrenoString[i][j] + "  ");
                }
            }
        }
        System.out.println("|");
    }
}
```

Este método muestra el terreno con la posición del tractor.

Generar problema

En esta clase se crea y se resuelven los problemas, además escribe la solución en un archivo de texto.

```
public Nodo RealizarBusqueda(int estrategia, int profundidadMaxima) {
    /**
     * Metodo que realiza las busquedas dependiendo de la estrategia
     */
    Scanner TECLADO = new Scanner(System.in);
    Nodo padre = null;
    Map<String, Integer> codigos = new HashMap<>();
    Frontera frontera = new Frontera();
    padre = new Nodo(0, 0, estadoInicial, null, null, 0);
    Nodo nodoComparar = null;
    Serializar serial = new Serializar();
    boolean solucion = false;
    int valor = 0;
    int cota = 0;
    int profundidadAcotada = 0;
    try {
        if (estrategia == 2) {
            System.out.println("Introduzca la cota máxima");
            cota = TECLADO.nextInt();
            profundidadAcotada=cota;
            if (cota < 0) {
                System.out.println("Error valor de entrada");
                RealizarBusqueda(estrategia, profundidadMaxima);
            }
        }
    } catch (Exception e) {
        System.out.println("Error formato de entrada");
        System.out.println(e.toString());
        RealizarBusqueda(estrategia, profundidadMaxima);
    }

    try {
        String id = serial.encriptarMD5(estadoInicial.convertirCadena());
        codigos.put(id, padre.getValor());
        frontera.insertarNodo(padre);
        if (estrategia == 5) {
            padre.setValor(estadoInicial.CalculoHeuristica());
        }
        while (!solucion && !frontera.esVacia()) {
            ArrayList<Sucesor> listaSucesores = new ArrayList<Sucesor>();
            ArrayList<Accion> listaAcciones = new ArrayList<Accion>();
            nodoComparar = frontera.eliminarNodo();
            if (nodoComparar.getEstado().isRepartido()) {
                solucion = true;
            } else if (nodoComparar.getProfundidad() <= profundidadMaxima) {
                listaAcciones = espacioEstados.obtenerAcciones(nodoComparar);
                listaSucesores = espacioEstados.obternerSucesores(listaAcciones, nodoComparar);
                for (int i = 0; i < listaSucesores.size(); i++) {
                    switch (estrategia) {
                    case 1:
                        // profuncidad simple
                        valor = (profundidadMaxima - (nodoComparar.getProfundidad() + 1) * (-1));
                        break;
                    case 2:
                        // profuncidad acotada
                        if (profundidadAcotada <= profundidadMaxima) {
                            valor = (profundidadMaxima - (nodoComparar.getProfundidad() + 1) * (-1));
```

```
                    } else {
                        frontera.limpiar();
                        frontera.insertarNodo(padre);
                        cota = cota + cota;
                    }
                    break;
                case 3:
                    // anchura
                    valor = nodoComparar.getProfundidad() + 1;
                    break;
                case 4:
                    // costo uniforme
                    valor = listaSucesores.get(i).getCoste();
                    break;
                case 5:
                    // asterisco
                    valor = listaSucesores.get(i).getCoste()
                            + listaSucesores.get(i).getEstado().CalculoHeuristica();
                    break;
                }
                Nodo nodo = new Nodo(nodoComparar.getProfundidad() + 1, listaSucesores.get(i).getCoste(),
                        listaSucesores.get(i).getEstado(), nodoComparar, listaSucesores.get(i).getAccion(),
                        valor);
                id = serial.encriptarMD5(nodo.getEstado().convertirCadena());
                // comprueba si el id creado esta en la lista, si esta se prueba que tenga un
                // valor menor el nodo, si es asi se mete
                // si el id no esta en la lista se mete
                if (codigos.containsKey(id)) {
                    if (nodo.getValor() < codigos.get(id)) {
                        codigos.replace(id, nodo.getValor());
                        frontera.insertarNodo(nodo);
                    }
                } else {
                    codigos.put(id, nodo.getValor());
                    frontera.insertarNodo(nodo);
                }
            }

        }
    }

} catch (Exception e) {
    System.out.println("Realizando busqueda error");
    System.out.println(e.getMessage());
    System.out.println(e.toString());
}
if (solucion) {
    System.out.println("***Existe solución***");
    return nodoComparar;
} else {
    System.out.println("***No existe solución***");
    return null;
}
```

Este método se encarga de crear el árbol de búsqueda, el método recibe una estrategia y la profundidad máxima del árbol, si supera esa profundidad, no hay solución.

El método encripta en MD5 para poder controlar los nodos iguales y podarlos, que consiste en si un nodo ya ha sido genera y tiene un valor superior a otro nodo con su misma encriptación, se deja de buscar en esa rama.

```
public ArrayList<Nodo> resultado(Nodo nodo) {
    /**
     * Crea el camino de nodos
     */
    ArrayList<Nodo> caminoNodos = new ArrayList<Nodo>();
    caminoNodos.add(nodo);
    while (nodo.getPadre() != null) {
        caminoNodos.add(nodo.getPadre());
        nodo = nodo.getPadre();
    }
    return caminoNodos;
}
```

Este método crea el camino de nodos solución

```
public void mostarSolucion(ArrayList<Nodo> nodoCamino) {
    /**
     *Muestro la sulucion obtenida y la escribo en un fichero
     */
    System.out.println("***Mostrando solución***");
    Collections.reverse(nodoCamino);
    fichero.BorrarFicheroSolucion();
    for (int i = 0; i < nodoCamino.size(); i++) {
        String[][] TerrenoString = null;
        TerrenoString = estados.CopiarTerreno2(nodoCamino.get(i).getEstado());
        estados.MostrarTerrenoTractor(nodoCamino.get(i).getEstado().getPosX(),
                nodoCamino.get(i).getEstado().getPosY(), TerrenoString);
        System.out.println("COSTO: " + nodoCamino.get(i).getCosto());
        System.out.println("PROF: " + nodoCamino.get(i).getProfundidad());
        System.out.println("VALOR: " + nodoCamino.get(i).getValor());
        System.out.println(nodoCamino.get(i).getAccion());
        fichero.EscrituraSolucionFichero(TerrenoString, nodoCamino.get(i).getEstado().getX(),
                nodoCamino.get(i).getEstado().getY(), nodoCamino.get(i).getEstado().getK(),
                nodoCamino.get(i).getEstado().getMax(), nodoCamino.get(i).getEstado().getPosX(),
                nodoCamino.get(i).getEstado().getPosY());
    }
}
```

Este método escribe la solución en un fichero y muestra por pantalla cada nodo solución.

Agente

En esta clase se crea el terreno inicial, junto con el estado inicial y se crea el problema.

Se pide la forma de crear el terreno, lectura por fichero o generar los datos de manera aleatoria

Métodos

```
public void Operaciones() {
    /**
     * En este metodo dependiendo de la forma de lectura del fichero se crea un
     * terreno con los parámetros leidos en un metedo GenerarEstados o generarlos de
     * manera aleatoria
     */
    ArrayList<Integer> contenido = new ArrayList<Integer>();
    Ficheros lecturaEscritura = new Ficheros();
    lecturaEscritura.BorrarFicheroSolucion();
    Posicion[][] terreno = null;
    Estado estado = null;
    GenerarTerreno generado = null;
    long tb0=0;
    long tb2=0;
    switch (PeticionFormatoEntradaDatos()) {
    case 1:
        System.out.println("--> Datos leidos por fichero");
        contenido = lecturaEscritura.LecturaFicheroTerrenoPrefijado();
        generado = new GenerarTerreno(contenido);
        terreno = generado.GenerarElTerreno();
        estado = new Estado(contenido, terreno, 0, 0);
        problema = new GenerarProblema(espacioEstados, estado, contenido);
        tb0 = System.nanoTime();
        MenusEstrategias(problema, contenido);
        tb2 = System.nanoTime();
        break;
    case 2:
        System.out.println("--> Datos generados aleatoriamente");
        int X = (int) (Math.random() * 2) + 3;
        contenido.add(X);
        System.out.println("    Número de filas del terreno: " + X);
        int Y=X;
        contenido.add(Y);
        System.out.println("    Número de columnas del terreno: " + Y);
        int maximo = (int) (Math.random() * 7) + 1;
        contenido.add(maximo);
        System.out.println("    Valor de tierra máximo en el terreno: " + maximo);
        int k = (int) (Math.random() * (maximo - 1)) + 1;
        contenido.add(k);
        System.out.println("    Valor de tierra mínimo en el terreno: " + k);
        int v = X * Y * k;
        contenido.add(v+1);
        System.out.println("    Tierra total del terreno: " + v);
        generado = new GenerarTerreno(contenido);
        terreno = generado.GenerarElTerreno();
        estado = new Estado(contenido, terreno, 0, 0);
        problema = new GenerarProblema(espacioEstados, estado, contenido);
        tb0 = System.nanoTime();
        MenusEstrategias(problema, contenido);
        tb2 = System.nanoTime();
        break;
    }
    System.out.println("El tiempo tardado es: " + (tb2 - tb0) + " Nanosegundos");
}
```

En este método se crea el terreno inicial y el estado inicial y se contabiliza el tiempo de ejecución del programa.

```
int profundidadMaxima = 9999;
Nodo nodoSolucion = null;
boolean solucion = false;
ArrayList<Nodo> nodoCaminos = null;
int opcion = 0;
while (!solucion) {
    try {
        System.out.println("----Estrategias disponibles----");
        System.out.println(
                " 1-Prondidad Simple \n 2-Profundidad Acotada \n 3-Anchura \n 4-Costo Uniforme \n 5-Asterisco");
        opcion = TECLADO.nextInt();
        switch (opcion) {
        case 1:
            System.out.println("---Profundidad Simple---");
            solucion = true;
            nodoSolucion = problema.RealizarBusqueda(1, profundidadMaxima);
            break;
        case 2:
            System.out.println("---Profundidad acotada---");
            solucion = true;
            nodoSolucion = problema.RealizarBusqueda(2, profundidadMaxima);
            break;
        case 3:
            System.out.println("---Anchura---");
            solucion = true;
            nodoSolucion = problema.RealizarBusqueda(3, profundidadMaxima);
            break;
        case 4:
            System.out.println("---Costo Uniforme---");
            solucion = true;
            nodoSolucion = problema.RealizarBusqueda(4, profundidadMaxima);
            break;
        case 5:
            System.out.println("---Asterisco---");
            solucion = true;
            nodoSolucion = problema.RealizarBusqueda(5, profundidadMaxima);
            break;
        default:
            System.out.println(opcion);
            System.out.println("**Error con los valores de entrada, parámetos permitidos 1, 2, 3, 4, 5**\n");
            MenusEstrategias(problema, contenido);
            break;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
        System.out.println("**Error con los valores de entrada, parámetos permitidos 1, 2, 3, 4, 5**\n");
        MenusEstrategias(problema, contenido);
    }
}
if (nodoSolucion == null) {
    System.out.println("***---Cota limite superado o sin solucion ---***");
} else {
    nodoCaminos = problema.resultado(nodoSolucion);
    problema.mostarSolucion(nodoCaminos);
}
```

En este método se selección la estrategia de resolución del problema.