PRACTICA 1

Allanar un terreno

Grupo BC1_09
Bautista Ruiz, Guillermo
Ramos López, Raquel
Vigara Arcos, Juan José

Contenido

Enlace github: https://github.com/ju4nko/BC1_09

Tarea 1

Trataremos de implementar un programa que solucione el problema de allanar un terreno con arena, encontrando la solución mediante técnicas de búsqueda. Para comenzar tendremos que elegir el lenguaje a utilizar y como vamos a estructurar el código, así como a realizar una vista previa del problema.

1.- Elecciones previas.

Como lenguaje hemos escogido Java, pues es el lenguaje que mejor conocemos los miembros del grupo. También nos inclinamos por usar *Neatbeans* como entorno de programación.

Usaremos como estructura un modelo de 3 capas dividas en dominio (donde estará la mayoría del código del programa), persistencia donde tenemos métodos que hemos preparado externamente como el tratamiento de los ficheros y utilidades donde tenemos métodos que vienen ya preestablecidos en java y que usamos en el programa.

2.- Definición del artefacto terreno.

Hemos creado una clase terreno que almacene los atributos de éste, y nos permita realizar operaciones sobre él, ya sea alguno que introduzcamos por teclado/fichero o generado aleatoriamente. En nuestro proyecto nos hemos decidido por un Menú en el que se da la opción al usuario de introducir los datos del terreno por teclado o indicando una ruta a un archivo con esos datos.

El terreno se compone de una matriz de objetos tipo Casilla de tamaño CxF, donde C es el número de columnas y F el número de filas. En la misma clase hemos creado sus métodos como por ejemplo "estaDentroAdyacente" que comprueba si el teórico adyacente de una casilla está en el rango del terreno (0, C-1) (0, F-1) además de los métodos típicos de cada clase como get, to string etc

3- Creación del artefacto terreno.

Para el problema podemos crear el artefacto terreno de varias maneras como hemos expresado anteriormente, tras crearlo se comprobará que se puede allanar (es decir que la cantidad de arena que tiene que estar en cada casilla concuerda con la cantidad total de arena y casillas).

3.1- Creación del terreno aleatorio.

El método ***rellenarTerrenoAleatorio(min, max)*** es llamado por el método ***crearTerreno()*** en el constructor de la clase Terreno. Éste llena de objetos de la clase Casilla una matriz con todas las casillas que va a tener nuestro terreno.

```
public void crearTerreno() {
    casillas = rellenarTerrenoAleatorio(0, MAX);
    comprobarAleatorio();
}
```

```
public Casilla[][] rellenarTerrenoAleatorio(int min, int max)
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            casillas[i][j] = new Casilla(i, j);
        }
    }
    return casillas;
}
```

Al crear en cada posición del array una casilla nueva, llamamos al método ***genAleatorioArena(min,max)*** de la clase Casilla en su constructor, que calculara aleatoriamente la cantidad de arena de cada casilla dentro de unos márgenes.

```
public int genAleatorioArena(int min,int max){
    return (int)(Math.random()*max+min);
}
```

Posteriormente, tenemos que comprobar que el total de arena creada aleatoriamente corresponde al total de arena que tiene que haber siguiendo la fórmula dada en el enunciado.

```
public void comprobarAleatorio() {
    int suma=0;
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            suma=+ casillas[i][j].getCantArena();
        }
    }
    if(suma>(filas*columnas*k)){
        int sobra;
        sobra=suma-(filas*columnas*k);
        for (int i = 0; i < filas && sobra!=0; i++) {
            for (int j = 0; j < columnas && sobra!=0; j++) {
                int valorC=casillas[i][j].getCantArena();
                while(valorC >0 && valorC<MAX && sobra!=0){
                    valorC--;
                    sobra--;
                    casillas[i][j].setCantArena(valorC);
                }

            }
        }
    }

    else if(suma<(filas*columnas*k)){
        int falta;
        falta=(filas*columnas*k)-suma;
        for (int i = 0; i < filas && falta!=0; i++) {
            for (int j = 0; j < columnas && falta!=0; j++) {
                int valorC=casillas[i][j].getCantArena();
                while(valorC >0 && valorC<MAX && falta!=0){
                    valorC++;
                    falta--;
                    casillas[i][j].setCantArena(valorC);
                }

            }
        }
    }
}
```

5	5	1
5	5	4
3	5	2
5	5	4

Ejemplo:
suma = 49
k= 4
filas = 4
columnas = 3
cantidadDeseada=4*4*3 =48
sobra 1 unidad de tierra ya colocada

buscamos posiciones que no lleguen al máximo de tierra para quitar todo lo que podamos y decrementamos la variable sobra, si sobra =0 ya tenemos nuestra disposición aleatoria, pero si sobra!=0 pasamos a la siguiente posición

3	5	1
5	2	4
3	5	2
5	5	4

Ejemplo:
suma = 44
k= 4
filas = 4
columnas = 3
cantidadDeseada=4*4*3 =48
faltan por colocar 4 unidades de tierra

buscamos posiciones que no lleguen al máximo de tierra para añadir todo lo que podamos y decrementamos la variable falta, si falta=0 ya tenemos nuestra disposición aleatoria, pero si falta!=0 pasamos a la siguiente posición

4.- Obtención de un nuevo terreno tras aplicar una acción.

Este programa funcionará de tal manera que tras cada acción se creará un nuevo terreno y se trabajará sobre ese en lugar de sobre el actual, el nuevo terreno será una copia del actual con las casillas afectadas cambiadas y la posición del tractor (las casillas afectadas son la casilla de la posición previa del tractor y sus adyacentes).

5.- Acciones del tractor.

Tenemos **dos** métodos principales para simular la acción de distribuir arena del tractor:

1.

```
public ArrayList<Casilla> accionTractor() {
    int i, j;
    Casilla aux;
    //Casilla posTractor = new Casilla(x,y); // Obtenemos la casilla donde está el tractor
    Casilla posTractor = getCasilla(t.getX(), t.getY());
    System.out.println("COORDENADAS: " + t.getX() + " " + t.getY());

    PonerVisitado(posTractor);
    ArrayList<Casilla> listaAdyacentes = new ArrayList();
    for (i = -1; i <= 1; i++) { // Todos los adyacentes de la casilla
        for (j = -1; j <= 1; j++) {
            //Obtenemos la posición en el terreno de los adyacentes
            aux = new Casilla(t.getX() + i, t.getY() + j);
            //aux = getCasilla(0+i, 0+j);
            if (estaDentro(aux)) {
                if (!EstaVisitado(aux)) {
                    aux = getCasilla(t.getX() + i, t.getY() + j);
                    if ((Math.abs(i) + Math.abs(j)) != 2) { // Cogemos los adyacentes que no sean diagonales
                        listaAdyacentes.add(aux);
                    }
                }
            }
        }
    }
    return listaAdyacentes;
}
```

Este método **acciónTractor** te calcula todos los adyacentes que tiene el tractor con respecto a su posición en el terreno. Cabe destacar que para quitar los adyacentes diagonales se le suma las coordenadas absolutas; en todos los casos la suma de las coordenadas absolutas de las casillas que están en diagonal siempre va a darnos 2, por lo tanto, quitamos estas casillas y no las metemos a la lista.

El método está dentro comprueba si una casilla está dentro de las dimensiones posibles de nuestro terreno. La casilla que esté fuera se obvia.

Por último el método **acciónTractor** devuelve una lista de objetos casilla que serán todas las casillas adyacentes al tractor.

2.

```java
public void backtracking(int etapa,int Sol[],int s,
                         ArrayList<Casilla> adyacentes,int MAX){
    int i;
    if(etapa==Sol.length){
        if(esSolucion(Sol,etapa,s)){
            // IMPRIMIMOS LAS COMBINACIONES POSIBLES DE DISTRIBUIR ARENA
            System.out.println(imprimirSol(Sol));
            // SUMAMOS LA ARENA A LA CASILLA A DISTRIBUIR
            imprimeLista(sumarListas(Sol,adyacentes,MAX));
            System.out.println();
        }
    }else{
        for(i=0;i<=s;i++){
            Sol[etapa] = i;
            backtracking(etapa+1,Sol,s,adyacentes,MAX);
        }
    }
}
```

Éste método es un backtracking que te hace todas las combinaciones de números posibles de números “s” en las posiciones del vector sol[]

El método **esSolucion** te filtra los resultados que sumando el vector[] sol se pasen de la cantidad “s”

Cuando tengamos una solución se la sumaremos a la lista que teníamos calculada en el vector de adyacentes y así obtendremos todas las posibles soluciones de distribución de arena.