Programación y Estructuras de Datos Avanzadas
Primera práctica

Carlos Caride Santeiro

1/12/2017

1. Datos personales

- Nombre y código de la asignatura: **Programación y Estructuras de Datos Avanzadas (71902019)**
- Título de la práctica: **Robot desplazándose en un circuito**
- Nombre y Apellidos: **Carlos Caride Santeiro**
- DNI: **44446239G**
- Centro asociado: **Ourense**

2. ENUNCIADO DE LA PRÁCTICA: Robot desplazándose en un circuito

Sea un robot R que dispone de una batería de N unidades de energía y se encuentra en un circuito por el que puede desplazarse. El objetivo es que R se desplace desde el punto en el que se encuentra hasta el punto de salida S del circuito, contando con que en el camino se puede encontrar con obstáculos, O, que son infranqueables. El paso por una casilla franqueable supone un gasto de energía igual al valor que indica la casilla, que deberá ser mayor que 0. Se busca un algoritmo que permita al robot llegar al punto S gastando el mínimo de energía.

El circuito se puede representar mediante una matriz de dimensiones *nxm* en la que desde cada elemento se puede acceder a un elemento adyacente con el consumo de energía que indique la casilla. En el apartado 3.8 del texto base se puede ver un ejemplo detallado de un circuito concreto. El esquema que se utilizará para su resolución será el indicado en el texto base para este problema: **esquema voraz**, en particular la solución basada en el algoritmo de Dijkstra.

3. Descripción del esquema algorítmico utilizado y como se aplica al problema

3.1. Descripción algoritmo de Dijkstra

Como se indica en el enunciado de la práctica, la solución se basa en el algoritmo de Dijkstra. El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de los vértices en un grafo con pesos en cada arista. Su nombre se refiere a *Edsger Dijkstra*, quien lo describió por primera vez en 1959.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. El algoritmo es una especialización de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Este algoritmo utiliza dos conjuntos de nodos, S y C. S contiene los nodos ya seleccionados y cuya distancia mínima al origen ya se conoce y C contiene los demás nodos, $C = N \setminus S$, aquellos cuya distancia mínima al origen no se conoce todavía. Al inicio del algoritmo S sólo contiene el nodo origen y cuando finaliza el algoritmo contiene todos los nodos del grafo y además se conocen las longitudes mínimas desde el origen a cada uno de ellos. La función de selección elegirá en cada paso el nodo de C cuya distancia al origen sea mínima.

El algoritmo de Dijkstra utiliza la noción de *camino especial*. Un camino desde el nodo origen hasta otro nodo es especial si todos los nodos intermedios del camino pertenecen a S, es decir, se conoce el camino mínimo desde el origen a cada uno de ellos. Hará falta un array, `especial[]`, que en cada paso del algoritmo contendrá la longitud del camino especial más corto (si el nodo está en S), o el camino más corto conocido (si el nodo está en C) que va desde el origen hasta cada nodo del grafo. Cuando se va a añadir un nodo a S, el camino especial más corto hasta ese nodo es también el más corto de todos los caminos posibles hasta él. Cuando finaliza el algoritmo todos los nodos están en S y por lo tanto todos los caminos desde el origen son caminos especiales.

Las longitudes o distancias mínimas que se esperan calcular con el algoritmo estarán almacenadas en el array `especial[]`.

Se supone que los nodos están numerados de 1 a n, por lo que $N = 1, 2, ..., n$, y que el nodo 1 es el nodo origen. También se supone que la función `Distancia()` devolverá la distancia o coste entre los dos nodos que son sus argumentos. Si existe una arista entre dichos nodos devolverá la etiqueta o peso asociado, en caso de que no exista una arista devolverá un valor representativo o suficientemente grande, por ejemplo ∞.

```
tipo VectorNat = matriz[O..n] de natural
fun Dijkstra (G = ⟨N,A⟩: grafo): VectorNat
    var
        especial: VectorNat
        C: conjunto de nodos
    fvar
    C={2,3,...,n}
    para i ← 2 hasta n hacer
        especial[i] ← Distancia(1,i)
    fpara
    mientras C contenga mas de 1 nodo hacer
        v ← nodo ∈ C que minimiza especial[v]
        C ← C\{v}
        para cada w ∈ C hacer
            especial[w] ← min(especial[w], especial[v] + Distancia(v,w))
        fpara
    fmientras
    dev especial[]
ffun
```

El bucle **mientras** se ejecuta $n - 2$ veces ya que cuando sólo queda un nodo en C no va a haber más modificaciones en el array `especial[]`. Si además de calcular el coste del camino mínimo desde el origen se quiere saber por dónde pasan los caminos, es necesario utilizar un array adicional, `predecesor[2 . . n]`, siendo `predecesor[i]` el identificador del nodo que precede al nodo i-ésimo en el camino más corto desde el origen. Añadiendo esta nueva funcionalidad el algoritmo quedaría:

```
tipo VectorNat = matriz[O..n] de natural
fun Dijkstra (G = ⟨N,A⟩: grafo): VectorNat, VectorNat
    var
```

```
        especial, predecesor: VectorNat
        C: conjunto de nodos
    fvar
    C={2,3,...,n}
    para i ← 2 hasta n hacer
        especial[i] ← Distancia(1 ,i)
        predecesor[i] ← 1
    fpara
    mientras C contenga mas de 1 nodo hacer
        v ← nodo ∈ C que minimiza especial[v]
        C ← C\{v}
        para cada w ∈ C hacer
            si especial[w] > especial[v] + Distancia(v,w) entonces
                especial[w] ← especial[v] + Distancia(v,w)
                predesecor[w] ← v
            fsi
        fpara
    fmientras
    dev especial[], predecesor[]
ffun
```

3.2. Aplicación al problema

El tablero o circuito se puede modelar como un grafo en el que cada casilla es un nodo y el contenido de la casilla el coste energético de acceder a dicho nodo por alguna de sus aristas incidentes. El grafo resultante es dirigido y por lo tanto la matriz no es simétrica.

Para un tablero de 5×5, los nodos se han numerado del 1 al 25 empezando por la posición $(1, 1)$ del tablero, siguiendo por la $(1, 2)$, $(1, 3)$,...hasta la $(5, 5)$. Dada una posición (i, j) del tablero, le corresponde el número de nodo $(i - 1) \times 5 + j$. En la posición del tablero en el que está el robot R, el coste de acceder a ella desde un nodo que no sea un obstáculo es 0, ya que no se indica otro posible coste, lo mismo ocurre con la posición S.

Con esta representación, el problema se reduce a encontrar un camino de coste mínimo desde la posición en la que se encuentra el robot, R, hasta la posición S. Este problema se puede solucionar utilizando el algoritmo de Dijkstra para calcular la longitud del camino mínimo que va desde el origen hasta el resto de los nodos del grafo. En este caso, el algoritmo Dijkstra se detendrá una vez que el camino hasta el nodo S ya se haya calculado, obteniendo así la ruta de coste mínimo en el camino de R a S.

El algoritmo de Dijkstra adaptado a este problema es el siguiente:

```
tipo VectorNat = matriz[O..n] de natural
fun Dijkstra (G = ⟨N,A⟩: grafo, R: natural, S: natural): VectorNat, VectorNat
    var
        especial, predecesor: VectorNat
        C: conjunto de nodos
    fvar
    C={1,2,3,...,n} excepto R
    para i ← 1 hasta n ∧ i ≠ R hacer
        especial[i] ← Distancia(R ,i)
        predecesor[i] ← R
    fpara
    mientras C contenga al nodo S hacer
        v ← nodo ∈ C que minimiza especial[v]
        C ← C\{v}
        si v ≠ S entonces
```

```
            para cada w ∈ C hacer
               si especial[w] > especial[v] + Distancia(v,w) entonces
                  especial[w] ← especial[v] + Distancia(v,w)
                  predesecor[w] ← v
               fsi
            fpara
         fsi
      fmientras
      dev especial[], predecesor[]
ffun
```

4. Demostración de optimalidad

La demostración de que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo se realiza por inducción. Se trata de demostrar que:

1. Si un nodo $i \neq 1$ está en S, entonces `especial[i]` almacena la longitud del camino más corto desde el origen, hasta el nodo `i`.
2. Si un nodo `i` no está en S, entonces `especial[i]` almacena la longitud del camino especial más corto desde el origen hasta el nodo `i`.

Además, por hipótesis estos dos puntos se cumplen inmediatamente antes de añadir un nodo v al conjunto S.

Base: inicialmente sólo el nodo inicial esta en S, por lo que el punto inicial está demostrado. Para el resto de los nodos el único camino especial posible es el camino directo al nodo inicial, cuya distancia es la que le asigna el algoritmo a `especial[i]`, por lo que el segundo punto también está demostrado.

Paso inductivo: como los nodos que ya están en S no se vuelven a examinar, el punto inicial se sigue cumpliendo antes de añadir un nodo v a S. Antes de poder añadir el nodo v a S hay que comprobar que `especial[v]` almacene la longitud del camino más corto o de menor coste desde el origen hasta v. Por hipótesis, `especial[v]` almacena la longitud del camino especial más corto, por lo que hay que verificar que dicho camino no pase por ningún nodo que no pertenece a S. Suponiendo que en el camino más corto desde el origen a v hay uno o más nodos, que no son v, que no pertenecen a S. Sea x el primer nodo de este tipo. Como x no está en S, `especial[x]` almacena la longitud del camino especial desde el origen hasta x. Como el algoritmo ha seleccionado a v antes que a x, `especial[x]` no es menor que `especial[v]`. Por lo tanto, la distancia total hasta v a través de x es como mínimo `especial[v]`, y el camino a través de x no puede ser más corto que el camino especial que lleva a v. Por lo tanto, cuando se añade v a S se cumple el punto 1.

Con respecto al punto 2, considerando un nodo u que no sea v y que no pertenece a S. Cuando se añade v a S puede darse una de dos posibles situaciones: (1) `especial[u]` no cambia porque no se encuentra un camino más corto a través de v o, (2) `especial[u]` si cambia porque se encuentra un camino más corto a través de v y quizás de algún otro u otros nodos de S. En este caso (2), sea x el último nodo de S visitado antes de llegar a u. La longitud de ese camino será `especial[x] + Distancia(x,u)`. Se podría pensar que para calcular el nuevo valor de `especial[u]` habría que comparar el valor anterior de `especial[u]` con los valores de `especial[x] + Distancia(x,u)` para todo $x \in S$, incluyendo a v. Pero esa comparación ya se hizo cuando se añadió x a S por lo que `especial[x]` no ha cambiado desde entonces. Por lo tanto, el cálculo del nuevo valor de `especial[u]` se puede hacer comparando solamente su valor anterior con `especial[v] + Distancia(v,u)`. Como esto es lo que hace el algoritmo, el punto 2 también se cumple cuando se añade un nuevo nodo v a S.

Al finalizar el algoritmo todos los nodos excepto uno estarán en S y el camino más corto desde el origen hasta dicho nodo es un camino especial. Por lo que queda demostrado que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo.

5. Coste computacional del algoritmo utilizado

En cuanto al coste, sea $G = \langle N, A \rangle$ un grafo, con n nodos y a aristas. Las tareas de inicialización están en $O(n)$. La selección de `v` dentro del bucle `mientras` requerirá examinar $n-1$, $n-2$,..., 2 valores de `especial[]` en los sucesivos pasos. Como el bucle `para` interno se ejecutará $n-2$, $n-3$,..., 1 veces, el tiempo requerido por el algoritmo de Dijkstra está en $O(n^2)$.

6. Alternativas al esquema utilizado

El coste computacional se puede reducir si se consigue evitar examinar todo el array `especial[]` cada vez que se quiere asignar a `v` el nodo de coste mínimo. Para ello, se podría utilizar un montículo de mínimos que almacenara los nodos de los que aún no se ha determinado su camino de coste mínimo desde el origen. Es decir, el montículo contendría los nodos de C, y en la raíz estaría el que minimiza el valor de `especial[]` . La inicialización del montículo está en $O(n)$. La instrucción que elimina v del conjunto e se traduce en eliminar la raíz del montículo, que está en $O(\log n)$. Si encontramos un nodo w tal que a través de v encontramos un camino menos costoso, debemos actualizar el montículo con w y ubicarlo según su valor de `especial[w]`, lo que requiere un tiempo que está en $O(\log n)$. Esto se hace como mucho una vez por cada arista del grafo. Por ello, se elimina la raíz del montículo $n-2$ veces y se realizan operaciones flotar un máximo de a veces, lo que da un tiempo que está en $O((n+a)\log n)$. Si el grafo es conexo, $n-1 \leq a \leq n^2$. Si el grafo es disperso, el número de aristas es pequeño y cercano a n, y la implementación con montículo está en $O(a \log n)$. Si el grafo es denso, el coste pasa a estar en $O(n^2 \log n)$ y la implementación sin montículo es preferible.

En el caso del problema, el uso de montículos seria recomendable, dado que $n-1 \leq a \leq n^2$ con un coste de para el caso de un mapa de $m \times n$. Se tiene un total de $m \times n$ nodos y un máximo de $4 \times 3 + 2(m-2) + 2(n-2) + 8(n-2)(m-2)$ aristas. En la tabla 1 se indica el coste de diferentes tamaños de mapa. Se puede comprobar que, a mayor tamaño de mapa, el uso de montículos se hace más recomendable.

Núm. filas	**Núm. col.**	**Núm. vért.**	**Núm. max. ar.**	**Sin mont.** $O(n^2)$	**Con mont.** $O((n+a)\log n)$
5	5	25	96	625	169
10	10	100	556	10.000	1.312
20	20	400	2.676	160.000	8.004
50	50	2.500	18.636	6.250.000	71.818

Tabla 1: Costes para diferentes valores de mapas

7. Datos de prueba

Para la comprobación del algoritmo se realizan 5 pruebas.

7.1. Prueba 1

7.1.1. Circuito

Circuito de ejemplo (ejemplo del apartado 3.8 del texto base):

O	S	O	2	1
3	1	3	1	1
1	6	6	O	6
1	2	O	R	4
7	1	1	2	6

7.1.2. Resultado obtenido

```
R[4,4],[5,3],[5,2],[4,1],[3,1],[2,2],S[1,2]

Energía total consumida: 5
```

7.2. Prueba 2

7.2.1. Circuito

O	1	O	O	O
1	O	1	5	5
S	O	1	5	5
5	O	1	5	R
5	O	5	1	5

7.2.2. Resultado obtenido

```
R[4,5],[5,4],[4,3],[3,3],[2,3],[1,2],[2,1],S[3,1]

Energía total consumida: 6
```

7.3. Prueba 3

7.3.1. Circuito

O	1	O	O	O
1	O	1	5	5
S	O	1	5	5
5	O	1	5	R
5	5	5	1	5

7.3.2. Resultado obtenido

```
R[4,5],[5,4],[4,3],[3,3],[2,3],[1,2],[2,1],S[3,1]

Energía total consumida: 6
```

7.4. Prueba 4

7.4.1. Circuito

O	O	O	O	O
1	O	1	5	5
S	O	1	5	5
5	O	1	5	R
5	5	5	1	5

7.4.2. Resultado obtenido

```
R[4,5],[5,4],[4,3],[5,2],[4,1],S[3,1]

Energía total consumida: 12
```

7.5. Prueba 5

7.5.1. Circuito

1	1	1	1	1
O	O	1	5	5
S	O	1	5	5
O	O	1	5	R
5	5	5	1	5

7.5.2. Resultado obtenido

Se obtiene excepción por no ser un circuito que se pueda resolver con los datos introducidos.

7.6. Resultado las pruebas

A la vista de los resultados obtenidos por el algoritmo, se estima que este es correcto. Se han probado casos de una única solución, así como de otros con varias o ninguna, obteniéndose siempre el camino mas corto.

8. ANEXO: Código fuente

Listado de archivos:

8.1. robot.Casilla

```
1 /*
2  * Archivo: Casilla.java
3  * Autor: Carlos Caride Santeiro
```

```
 * DNI: 44446239G
 * Email: ccaride5@alumno.uned.es
 * Fecha: 01/12/2017
 * Asignatura: Programación y Estructuras de Datos Avanzadas.
 * Trabajo: Práctica 1 (2017-2018)
 */

package robot;

/**
 * Representa una casilla del circuito
 */
public class Casilla {
    /**
     * Valor infinito
     */
    public static final int COSTE_INFINITO = 99999;

    private TipoCasilla tipo;
    private int valorCasilla;

    /**
     * Retorna el tipo de casilla
     * @return el tipo de casilla
     */
    public TipoCasilla getTipo() {
        return tipo;
    }

    /**
     * Establece el tipo de casilla
     * @param tipo el tipo a establecer
     */
    public void setTipo(TipoCasilla tipo) {
        this.tipo = tipo;
    }

    /**
     * Retorna el valor de la casilla
     * @return el valor de la casilla
     */
    public int getValorCasilla() {
        return valorCasilla;
    }

    /**
     * Establece el valor de la casilla
     * @param valorCasilla el valor de la casilla
     */
    public void setValorCasilla(int valorCasilla) {
        this.valorCasilla = valorCasilla;
    }

    /**
     * Constructor genérico
     */
```

```
60      public Casilla(){
61          this(TipoCasilla.NoAccesible, COSTE_INFINITO);
62      }
63
64      /**
65       * Constructor de una nueva casilla indicando el tipo y coste
66       * @param tipo el tipo de casilla
67       * @param coste el coste de atravesar la casilla
68       */
69      public Casilla(TipoCasilla tipo, int coste){
70          this.tipo = tipo;
71          valorCasilla = coste;
72      }
73  }
```

8.2. robot.Circuito

```
1   /*
2    * Archivo: Circuito.java
3    * Autor: Carlos Caride Santeiro
4    * DNI: 44446239G
5    * Email: ccaride5@alumno.uned.es
6    * Fecha: 01/12/2017
7    * Asignatura: Programación y Estructuras de Datos Avanzadas.
8    * Trabajo: Práctica 1 (2017-2018)
9    */
10
11  package robot;
12
13  import java.io.BufferedReader;
14  import java.io.File;
15  import java.io.FileNotFoundException;
16  import java.io.FileReader;
17  import java.io.IOException;
18  import java.io.PrintStream;
19  import java.util.ArrayList;
20  import robot.grafos.Grafo;
21  import robot.grafos.voraces.Dijkstra;
22
23  /**
24   * Representa el circuito que se tiene que resolver
25   */
26  public class Circuito {
27
28      private int numeroFilas;
29      private int numeroColumnas;
30      private int posicionRobot;
31      private int posicionRobotColumna;
32      private int posicionRobotFila;
33      private int posicionSalida;
34      private int posicionSalidaColumna;
35      private int posicionSalidaFila;
36      private boolean traza;
37      private String ficheroEntrada;
38      private String ficheroSalida;
```

```
39      private Casilla[][] circuito;
40      private Grafo grafo;
41
42      /**
43       * Constructor genérico
44       */
45      public Circuito() {
46          this(false, null, null);
47      }
48
49      /**
50       * Constructor de circuito con ficheros de entrada y salida
51       * @param ficheroEntrada Fichero que tiene descrito el circuito
52       * @param ficheroSalida  Fichero donde se almacenará la solución
53       */
54      public Circuito(String ficheroEntrada, String ficheroSalida) {
55          this(false, ficheroEntrada, ficheroSalida);
56      }
57
58      /**
59       * Constructor de circuito con ficheros de entrada y salida con traza
60       * @param traza Verdadero si se desea imprimir la traza
61       * @param ficheroEntrada Fichero que tiene descrito el circuito
62       * @param ficheroSalida  Fichero donde se almacenará la solución
63       */
64      public Circuito(boolean traza, String ficheroEntrada,
65              String ficheroSalida) {
66          this.traza = traza;
67          this.ficheroEntrada = ficheroEntrada;
68          this.ficheroSalida = ficheroSalida;
69          posicionRobot = -1;
70          posicionRobotFila = -1;
71          posicionRobotColumna = -1;
72          posicionSalida = -1;
73          posicionSalidaColumna = -1;
74          posicionSalidaFila = -1;
75      }
76
77      /**
78       * Carga los datos correspondientes del circuito contenidos en el fichero
79       * @return Verdadero en caso de que esté todo correcto
80       */
81      public boolean cargarDatos() {
82          // Comprobamos que se indicara el nombre del fichero
83          if (ficheroEntrada.isEmpty()) {
84              System.out.println("ERROR: No se indicó fichero de entrada");
85              return false;
86          }
87          try {
88              FileReader f = new FileReader(ficheroEntrada);
89              BufferedReader b = new BufferedReader(f);
90              String cadena;
91              // Leemos la definición del cirucito (num. filas y columnas)
92              cadena = b.readLine();
93              numeroFilas = Integer.parseInt(cadena);
94              cadena = b.readLine();
```

```
95              numeroColumnas = Integer.parseInt(cadena);
96              circuito = new Casilla[numeroFilas][numeroColumnas];
97
98              // Comenzamos la lectura del fichero
99              for (int i = 0; i < numeroFilas; i++) {
100                 for (int j = 0; j < numeroColumnas; j++) {
101                     cadena = b.readLine();
102
103                     if (cadena.equalsIgnoreCase("R")) {
104                         if (posicionRobotFila != -1) {
105                             System.out.println("ERROR: Se encontró más de una "
106                                     + "posición inicial del robot");
107                             b.close();
108                             return false;
109                         }
110                         posicionRobotFila = i;
111                         posicionRobotColumna = j;
112                         circuito[i][j] = new Casilla(TipoCasilla.Robot, 0);
113                     } else if (cadena.equalsIgnoreCase("O")) {
114                         circuito[i][j] = new Casilla(TipoCasilla.Obstaculo,
115                                 Casilla.COSTE_INFINITO);
116                     } else if (cadena.equalsIgnoreCase("S")) {
117                         if (posicionSalidaFila != -1) {
118                             System.out.println("ERROR: Se encontró más de una "
119                                     + "salida");
120                             b.close();
121                             return false;
122                         }
123
124                         if (i != 0
125                                 && i != (numeroFilas - 1)
126                                 && j != 0
127                                 && j != (numeroColumnas - 1)) {
128                             System.out.println("ERROR: La salida no se "
129                                     + "encuentra en una casilla periférica");
130                             b.close();
131                             return false;
132                         }
133                         posicionSalidaFila = i;
134                         posicionSalidaColumna = j;
135
136                         circuito[i][j] = new Casilla(TipoCasilla.Salida, 0);
137                     } else {
138                         circuito[i][j] = new Casilla(TipoCasilla.Franqueable,
139                                 Integer.parseInt(cadena));
140                     }
141                 }
142             }
143             b.close();
144         } catch (FileNotFoundException ex) {
145             System.out.println("ERROR: No se encontró el fichero de entrada");
146             return false;
147         } catch (IOException | NumberFormatException ex) {
148             System.out.println("ERROR: El formato del fichero es erróneo");
149             return false;
150         }
```

```
151
152         // Comprobamos que haya un robot en el circuito y calculamos su nodo
153         if (posicionRobotFila == -1) {
154             System.out.println("ERROR: No se encontró posición inicial del"
155                     + " robot");
156             return false;
157         } else {
158             posicionRobot = posicionRobotFila * numeroFilas +
159                     posicionRobotColumna;
160         }
161
162         // Comprobamos que haya una salida en el circuito y calculamos su nodo
163         if (posicionSalidaFila == -1) {
164             System.out.println("ERROR: No se encontró ninguna salida");
165             return false;
166         } else {
167             posicionSalida = posicionSalidaFila * numeroFilas +
168                     posicionSalidaColumna;
169         }
170
171         // Creamos el grafo del circuito
172         crearGrafo();
173
174         return true;
175     }
176
177     /**
178      * Crea el grafo correspondiente al circuito leído
179      */
180     private void crearGrafo() {
181         grafo = new Grafo(numeroColumnas * numeroFilas);
182
183         // Se calcula el valor de cada casilla, así como los costes de las
184         // casillas adyacentes.
185         for (int i = 0; i < numeroFilas; i++) {
186             for (int j = 0; j < numeroColumnas; j++) {
187                 if (circuito[i][j].getTipo() == TipoCasilla.Obstaculo) {
188                     continue;
189                 }
190
191                 int verticeActual = i * numeroFilas + j;
192                 int valor = circuito[i][j].getValorCasilla();
193
194                 if (i != 0) {
195                     if (j != 0) {
196                         if (circuito[i - 1][j - 1].getTipo() !=
197                                 TipoCasilla.Obstaculo) {
198                             grafo.aniadirArista(verticeActual - numeroColumnas
199                                     - 1, verticeActual, valor);
200                         }
201                     }
202                     if (circuito[i - 1][j].getTipo() != TipoCasilla.Obstaculo) {
203                         grafo.aniadirArista(verticeActual - numeroColumnas,
204                                 verticeActual, valor);
205                     }
206                     if (j != (numeroColumnas - 1)) {
```

```
207                         if (circuito[i - 1][j + 1].getTipo() !=
208                                 TipoCasilla.Obstaculo) {
209                             grafo.aniadirArista(verticeActual - numeroColumnas
210                                     + 1, verticeActual, valor);
211                         }
212                     }
213                 }
214
215                 if (j != 0) {
216                     if (circuito[i][j - 1].getTipo() != TipoCasilla.Obstaculo) {
217                         grafo.aniadirArista(verticeActual - 1, verticeActual,
218                                 valor);
219                     }
220                 }
221                 if (j != (numeroColumnas - 1)) {
222                     if (circuito[i][j + 1].getTipo() != TipoCasilla.Obstaculo) {
223                         grafo.aniadirArista(verticeActual + 1, verticeActual,
224                                 valor);
225                     }
226                 }
227
228                 if (i < (numeroFilas - 1)) {
229                     if (j != 0) {
230                         if (circuito[i + 1][j - 1].getTipo() !=
231                                 TipoCasilla.Obstaculo) {
232                             grafo.aniadirArista(verticeActual + numeroColumnas
233                                     - 1, verticeActual, valor);
234                         }
235                     }
236                     if (circuito[i + 1][j].getTipo() != TipoCasilla.Obstaculo) {
237                         grafo.aniadirArista(verticeActual + numeroColumnas,
238                                 verticeActual, valor);
239                     }
240                     if (j != (numeroColumnas - 1)) {
241                         if (circuito[i + 1][j + 1].getTipo() !=
242                                 TipoCasilla.Obstaculo) {
243                             grafo.aniadirArista(verticeActual + numeroColumnas
244                                     + 1, verticeActual, valor);
245                         }
246                     }
247                 }
248             }
249         }
250     }
251
252     /**
253      * Resuelve el circuito mediante el algoritmo de Dijkstra
254      */
255     public void resolver() throws FileNotFoundException, IOException {
256         Dijkstra d;
257         PrintStream ps;
258         if (ficheroSalida.isEmpty()) {
259             ps = System.out;
260         } else {
261             File f = new File(ficheroSalida);
262             if (f.exists()) {
```

```
263                 f.delete();
264             }
265             f.createNewFile();
266             ps = new PrintStream(f);
267         }
268         if (!traza) {
269             d = new Dijkstra(grafo, posicionRobot, posicionSalida);
270         } else {
271             d = new Dijkstra(grafo, posicionRobot, posicionSalida, ps);
272             ps.println();
273         }
274         imprimirSolucion(d, ps);
275     }
276 
277     /**
278      * Imprime la solución que se ha obtenido
279      */
280     private void imprimirSolucion(Dijkstra d, PrintStream ps) {
281         int[] predecesor = d.getPredecesor();
282         int coste = d.getEspecial()[posicionSalida];
283         int posicion = posicionSalida;
284         ArrayList<Integer> nodos = new ArrayList<>();
285 
286         nodos.add(posicion);
287 
288         while(!nodos.contains(posicionRobot)) {
289             nodos.add(predecesor[posicion]);
290             posicion = predecesor[posicion];
291         }
292 
293         ps.print("R");
294         for (int i = nodos.size() - 1; i >= 0; i--) {
295             if (i== 0) {
296                 ps.print("S");
297             }
298             posicion = nodos.get(i);
299             ps.print("[");
300             ps.print((posicion / numeroColumnas) + 1);
301             ps.print(",");
302             ps.print((posicion % numeroColumnas) + 1);
303             ps.print("]");
304             if (i != 0) {
305                 ps.print(",");
306             } else {
307                 ps.println();
308             }
309         }
310         ps.print("Energía total consumida: ");
311         ps.println(coste);
312     }
313 }
```

8.3. robot.Robot

```java
/*
 * Archivo: Robot.java
 * Autor: Carlos Caride Santeiro
 * DNI: 44446239G
 * Email: ccaride5@alumno.uned.es
 * Fecha: 01/12/2017
 * Asignatura: Programación y Estructuras de Datos Avanzadas.
 * Trabajo: Práctica 1 (2017-2018)
 */

package robot;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Clase principal de la aplicación
 */
public class Robot {

    /**
     * Punto de entrada de la aplicación
     */
    public static void main(String[] args) {

        boolean traza = false;
        String ficheroIn;
        String ficheroOut = "";
        int desfaseArgumentos = 0;

        if (args.length == 0) {
            System.out.println("ERROR: Sintaxis incorrecta");
            ImprimirAyuda();
            return;
        }
        if (args[0].equalsIgnoreCase("-h")){
            ImprimirAyuda();
            return;
        }

        if (args[0].equalsIgnoreCase("-t")){
            traza = true;
            desfaseArgumentos++;
        }

        if (args.length <= desfaseArgumentos) {
            System.out.println("ERROR: Pocos argumentos");
            ImprimirAyuda();
            return;
        }

        ficheroIn = args[desfaseArgumentos++];

```

```
55          if (args.length > desfaseArgumentos) {
56              ficheroOut = args[desfaseArgumentos];
57          }
58
59          Circuito c = new Circuito(traza, ficheroIn, ficheroOut);
60          if (!c.cargarDatos()) {
61              return;
62          }
63
64          try {
65              c.resolver();
66          } catch (IOException ex) {
67              Logger.getLogger(Robot.class.getName()).log(Level.SEVERE, null, ex);
68          }
69      }
70
71      /**
72       * Imprime la ayuda que se le mostrará al usuario
73       */
74      private static void ImprimirAyuda() {
75          System.out.println("SINTAXIS:");
76          System.out.println("robot [-t][-h][fichero_entrada] [fichero_salida]");
77          System.out.println("-t               Traza la selección de clientes");
78          System.out.println("-h               Muestra esta ayuda");
79          System.out.println("fichero_entrada  Nombre del fichero de entrada");
80          System.out.println("fichero_salida   Nombre del fichero de salida");
81      }
82
83  }
```

8.4. robot.TipoCasilla

```
1   /*
2    * Archivo: TipoCasilla.java
3    * Autor: Carlos Caride Santeiro
4    * DNI: 44446239G
5    * Email: ccaride5@alumno.uned.es
6    * Fecha: 01/12/2017
7    * Asignatura: Programación y Estructuras de Datos Avanzadas.
8    * Trabajo: Práctica 1 (2017-2018)
9    */
10
11  package robot;
12
13  /**
14   * Tipo de casillas que pueden existir en el mapa
15   */
16  public enum TipoCasilla {
17      /**
18       * Casilla no accesible
19       */
20      NoAccesible,
21      /**
22       * Posición del robot
23       */
```

```
24      Robot,
25      /**
26       * Salida del mapa
27       */
28      Salida,
29      /**
30       * Obstáculo, casilla no franqueable
31       */
32      Obstaculo,
33      /**
34       * Casilla por la cual el robot puede pasar
35       */
36      Franqueable
37  }
```

8.5. robot.grafos.Arista

```
1   /*
2    * Archivo: Arista.java
3    * Autor: Carlos Caride Santeiro
4    * DNI: 44446239G
5    * Email: ccaride5@alumno.uned.es
6    * Fecha: 01/12/2017
7    * Asignatura: Programación y Estructuras de Datos Avanzadas.
8    * Trabajo: Práctica 1 (2017-2018)
9    */
10
11  package robot.grafos;
12
13  /**
14   * Representa una arista de un grado
15   */
16  public class Arista {
17      private int peso;
18
19      public Arista() {
20          this(0);
21      }
22
23      /**
24       * Constructor genérico
25       * @param peso
26       */
27      public Arista(int peso) {
28          this.peso = peso;
29      }
30
31      /**
32       * Obtiene el peso de la arista
33       * @return el peso de la arista
34       */
35      public int getPeso() {
36          return peso;
37      }
38
```

```
39      /**
40       * Establece el peso de la arista
41       * @param peso el peso de la arista
42       */
43      public void setPeso(int peso) {
44          this.peso = peso;
45      }
46  }
```

8.6. robot.grafos.Grafo

```
1   /*
2    * Archivo: Grafo.java
3    * Autor: Carlos Caride Santeiro
4    * DNI: 44446239G
5    * Email: ccaride5@alumno.uned.es
6    * Fecha: 01/12/2017
7    * Asignatura: Programación y Estructuras de Datos Avanzadas.
8    * Trabajo: Práctica 1 (2017-2018)
9    */
10
11  package robot.grafos;
12
13  import java.util.*;
14
15  /**
16   * Representa un grafo
17   */
18  public class Grafo {
19
20      int numeroNodos;
21      Arista[][] matrizAdyacencia;
22
23      /**
24       * Constructor genérico
25       */
26      public Grafo() {
27
28      }
29
30      /**
31       * Constructor de un nuevo grafo con en número de nodos indicado
32       * @param nNodos Número de nodos del grafo
33       */
34      public Grafo(int nNodos) {
35          numeroNodos = nNodos;
36          matrizAdyacencia = new Arista[nNodos][nNodos];
37      }
38
39      /**
40       * Añade una nueva arista al grafo
41       * @param nInicio Nodo de inicio
42       * @param nDestino Nodo de destino
43       * @param peso Peso de la arista
44       */
```

```
45      public void aniadirArista(int nInicio, int nDestino, int peso) {
46          if (nInicio >= numeroNodos || nDestino >= numeroNodos) {
47              throw new IndexOutOfBoundsException("Nodo de inicio/destino"
48                      + " no existe");
49          }
50          matrizAdyacencia[nInicio][nDestino] = new Arista(peso);
51      }
52
53      /**
54       * Borra la arista indicada
55       * @param nInicio Nodo de inicio de la arista
56       * @param nDestino Nodo de destino de la arista
57       */
58      public void borrarArista(int nInicio, int nDestino) {
59          if (nInicio >= numeroNodos || nDestino >= numeroNodos) {
60              throw new IndexOutOfBoundsException("Nodo de inicio/destino"
61                      + " no existe");
62          }
63          matrizAdyacencia[nInicio][nDestino] = null;
64      }
65
66      /**
67       * Indica si dos nodos son adyacentes
68       * @param n1 Primer nodo
69       * @param n2 Segundo nodo
70       * @return Devuelve verdadero en caso de ser adyacentes
71       */
72      public boolean esAdyacente(int n1, int n2) {
73          if (n1 >= numeroNodos || n2 >= numeroNodos) {
74              throw new IndexOutOfBoundsException("Nodo de inicio/destino"
75                      + " no existe");
76          }
77          return !(matrizAdyacencia[n1][n2] == null &&
78                  matrizAdyacencia[n2][n1] == null);
79      }
80
81      /**
82       * Devuelve una lista con los nodos adyacentes a uno dado
83       * @param nodo Nodo del que se quieren obtener los adyacentes
84       * @return Lista con los
85       */
86      public ArrayList<Integer> adyacentes(int nodo) {
87          if (nodo >= numeroNodos) {
88              throw new IndexOutOfBoundsException("Nodo de inicio/destino"
89                      + " no existe");
90          }
91          ArrayList<Integer> lista = new ArrayList<Integer>();
92          for (int i = 0; i < numeroNodos; i++) {
93              if (matrizAdyacencia[nodo][i] != null) {
94                  lista.add(i);
95              }
96          }
97          return lista;
98      }
99
100     /**
```

```
101         * Devuelve el peso de la arista de dos nodos adyacentes
102         * @param nInicio Nodo inicial
103         * @param nDestino Nodo destino
104         * @return El peso de la arista
105         */
106        public int peso(int nInicio, int nDestino) {
107            if (nInicio >= numeroNodos || nDestino >= numeroNodos) {
108                throw new IndexOutOfBoundsException("Nodo de inicio/destino"
109                        + " no existe");
110            }
111            if (matrizAdyacencia[nInicio][nDestino] != null) {
112                return matrizAdyacencia[nInicio][nDestino].getPeso();
113            } else {
114                throw new NoSuchElementException("No existe adyacente.");
115            }
116        }
117
118        /**
119         * Imprime la matriz de adyacencia del grafo
120         */
121        public void imprimir() {
122            for (int i = 0; i < numeroNodos; i++) {
123                if (i < 10) {
124                    System.out.print(" ");
125                }
126                System.out.print(i);
127                System.out.print(" ");
128            }
129            System.out.println();
130            for (int i = 0; i < numeroNodos; i++) {
131                for (int j = 0; j < numeroNodos; j++) {
132                    if (matrizAdyacencia[i][j] == null) {
133                        System.out.print(" \u221E ");
134                        continue;
135                    }
136                    System.out.print(" ");
137                    System.out.print(matrizAdyacencia[i][j].getPeso());
138                    System.out.print(" ");
139                }
140                System.out.println();
141            }
142        }
143
144        /**
145         * Obtiene el número de nodos del grafo
146         * @return Número de nodos
147         */
148        public int getNumeroNodos() {
149            return this.numeroNodos;
150        }
151    }
```

8.7. robot.grafos.voraces.Dijkstra

```
1  /*
2   * Archivo: Dijkstra.java
3   * Autor: Carlos Caride Santeiro
4   * DNI: 44446239G
5   * Email: ccaride5@alumno.uned.es
6   * Fecha: 01/12/2017
7   * Asignatura: Programación y Estructuras de Datos Avanzadas.
8   * Trabajo: Práctica 1 (2017-2018)
9   */
10
11 package robot.grafos.voraces;
12
13 import java.io.PrintStream;
14 import java.util.ArrayList;
15 import java.util.Iterator;
16 import robot.Casilla;
17 import robot.grafos.Grafo;
18
19 /**
20  * Implementación del algoritmo de Dijkstra de caminos mínimos
21  */
22 public class Dijkstra {
23     int especial[], predecesor[];
24
25     /**
26      * Resuelve el camino mínimo del grafo indicado
27      * @param g El grafo a resolver
28      * @param inicio Nodo de inicio
29      * @param fin Nodo final
30      */
31     public Dijkstra(Grafo g, int inicio, int fin) {
32         this(g, inicio, fin, null);
33     }
34
35     /**
36      * Resuelve el camino mínimo del grafo indicado trazando la solución
37      * @param g El grafo a resolver
38      * @param inicio Nodo de inicio
39      * @param fin Nodo final
40      * @param ps Stream de salida
41      */
42     public Dijkstra(Grafo g, int inicio, int fin, PrintStream ps) {
43         int paso = 0;
44         ArrayList<Integer> C = new ArrayList<>();
45         especial = new int[g.getNumeroNodos()];
46         predecesor = new int[g.getNumeroNodos()];
47         Integer v, w, distancia;
48
49         if (ps != null) {
50             ps.println("Paso\tv\tC\tespecial\tpredecesor");
51         }
52
53         for (int i = 0; i < g.getNumeroNodos(); i++) {
54             if (i == inicio) {
```

```
55                  especial[i] = Casilla.COSTE_INFINITO;
56                  continue;
57              }
58              C.add(i);
59
60              if (g.esAdyacente(inicio, i)) {
61                  especial[i] = g.peso(inicio, i);
62              } else {
63                  especial[i] = Casilla.COSTE_INFINITO;
64              }
65              predecesor[i] = inicio;
66          }
67
68          if (ps != null) {
69              ps.print("Ini.\t\t");
70              ps.print(C.get(0) + 1);
71              for (int i = 1; i < C.size(); i++) {
72                  ps.print(",");
73                  ps.print(C.get(i) + 1);
74              }
75              ps.print("\t");
76              imprimirEspecial(ps, inicio);
77              ps.print("\t");
78              imprimirPredecesor(ps, inicio);
79              ps.println();
80          }
81
82          while (C.contains(fin)) {
83              paso++;
84              v = minimoNodo(C);
85              C.remove(Integer.valueOf(v));
86              if (v == -1) {
87                  throw new Error("El circuito no se puede resolver");
88              }
89              if (v != fin) {
90                  for (Iterator<Integer> iter = C.iterator(); iter.hasNext();) {
91                      w = iter.next();
92
93                      if (g.esAdyacente(v, w)) {
94                          distancia = g.peso(v, w);
95                      } else {
96                          distancia = Casilla.COSTE_INFINITO;
97                      }
98
99                      if(especial[w] > (especial[v] + distancia)) {
100                         especial[w] = especial[v] + distancia;
101                         predecesor[w] = v;
102                     }
103                 }
104             }
105             if (ps != null) {
106                 ps.print(paso);
107                 ps.print("\t");
108                 ps.print(v + 1);
109                 ps.print("\tC \\{");
110                 ps.print(v + 1);
```

```
111                 ps.print("}\t");
112                 imprimirEspecial(ps, inicio);
113                 ps.print("\t");
114                 imprimirPredecesor(ps, inicio);
115                 ps.println();
116             }
117         }
118     }
119 
120     /**
121      * Calcula el nodo que minimiza especial
122      * @param C Lista de nodos a comprobar
123      * @return El nodo mínimo
124      */
125     private int minimoNodo(ArrayList<Integer> C) {
126         int minimo = Casilla.COSTE_INFINITO - 1;
127         int vertice = -1;
128         for (Iterator<Integer> iter = C.iterator(); iter.hasNext();) {
129             int i = iter.next();
130             if (minimo > especial[i]) {
131                 minimo = especial[i];
132                 vertice = i;
133             }
134         }
135         return vertice;
136     }
137 
138     /**
139      * Devuelve una matriz de los valores especial
140      * @return Matriz de los valores especial
141      */
142     public int[] getEspecial() {
143         return especial.clone();
144     }
145 
146     /**
147      * Devuelve una matriz de los nodos predecesores
148      * @return Matriz de los nodos predecesores
149      */
150     public int[] getPredecesor() {
151         return predecesor.clone();
152     }
153 
154     /**
155      * Imprime la matriz especial
156      * @param ps Stream de salida
157      * @param inicio Nodo de inicio del algoritmo
158      */
159     private void imprimirEspecial(PrintStream ps, int inicio) {
160         for (int i = 0; i < especial.length; i++) {
161             ps.print(" ");
162             if (i != inicio) {
163                 if (especial[i] == Casilla.COSTE_INFINITO) {
164                     ps.print("i");
165                 } else {
166                     ps.print(especial[i]);
```

```
167                 }
168             } else {
169                 ps.print("-");
170             }
171         }
172     }
173 
174     /**
175      * Imprime la matriz predecesor
176      * @param ps Stream de salida
177      * @param inicio Nodo de inicio del algoritmo
178      */
179     private void imprimirPredecesor(PrintStream ps, int inicio) {
180         for (int i = 0; i < predecesor.length; i++) {
181             ps.print(" ");
182             if (i != inicio) {
183                 ps.print(predecesor[i] + 1);
184             } else {
185                 ps.print("-");
186             }
187         }
188     }
189 }
```