Curso 2016-2017
Estructura de Datos y Algoritmia
Práctica 1 (27 de septiembre)

lsi Departamento de Lenguajes y Sistemas Informáticos

Objetivo

En esta primera práctica vamos a recordar cómo se puede extraer información a partir de ficheros de texto, cómo almacenar información en ficheros de texto y recordar el uso de estructuras de datos sencillas para realizar algunas aplicaciones.

En este caso, hablamos de leer información que nos permitirá reconstruir un plano de un laberinto.

Realizaremos:

- Implementación y uso de clases para gestionar la entrada y salida de información a partir de ficheros de texto.
- Implementación de estructuras de datos para el almacenamiento interno de la información leída desde fichero.
- Implementación de aplicaciones concretas con las clases definidas.

Plazo de entrega: desde el lunes 10 de octubre hasta el **viernes 14 de octubre**. Más información al final de este documento.

Gestión de información desde ficheros de texto

Para poder realizar las tareas que nos interesa, aparte de implementar estructuras de datos adecuadas, necesitamos en primer lugar leer la información almacenada en ficheros de texto con un formato concreto, información que finalmente será almacenada adecuadamente en dichas estructuras.

La información que tenemos podría ser un plano de una región o país del planeta (u otros planetas), en el que se nos indica las vías posibles de comunicación entre varios puntos de interés. Los puntos de interés pueden tener diferente naturaleza, y por tanto, diferente información asociada. Por un lado tendremos los espacios por los que nos podremos mover a lo largo del plano y por otro los posibles obstáculos que no nos permitirán avanzar. Entre los espacios accesibles, podemos encontrarnos distintas categorías, como son comienzo y destinos posibles, así como puertas de acceso con diferentes propiedades.

Tendremos almacenada información relevante de todo esto en un fichero de texto que tendremos que procesar y almacenar adecuadamente en estructuras de datos.

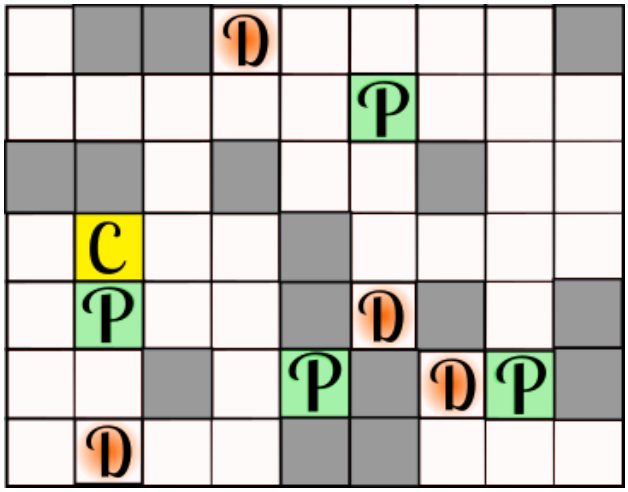
En primer lugar tendremos la información referente al plano indicando sólamente si las posiciones son libres ('l') o son obstáculos ('o'). Después vendrá la información asociada a destinos [1] y puertas con las siguientes características:

`<DESTINO>`

- nombre
- coordenadas en el mapa

`<PUERTA>`

- número de puerta
- coordenadas en el mapa
- propiedades de la puerta que será un carácter



[1] la información asociada a un comienzo es equivalente

Por ejemplo, parte de un fichero de texto para el mapa anterior sería el siguiente:

```
<MAPA>
loollllo
lllllllll
oololloll
llllollll
llllololo
llollollo
llllloolll
<DESTINO>
Linz
0 3
<DESTINO>
Graz
4 5
<PUERTA>
1
1 5
P
<PUERTA>                    //esta puerta es incorrecta porque hay un obstáculo
5
0 8
T
<PUERTA>            //esta puerta es incorrecta porque no hay propiedad definida
15
1 8
<PUERTA>                    //esta puerta es incorrecta porque no tiene código
1 8
C
<DESTINO>   //este destino es incorrecto porque no tiene nombre y es un obstáculo
2 3
<DESTINO>
Hallstatt
5 6
<COMIENZO>
Melk
3 1
...
```

Como se puede observar en el ejemplo, los diferentes tipos de casillas (comienzo,

destinos o puertas) pueden estar en cualquier orden. Puede haber además más de un comienzo, pero sólo se tendrá en cuenta el primero válido. También se puede observar que si hay errores en el formato se debe descartar la información relacionada (falte el nombre en los destinos o comienzo, falten las coordenadas, o falte la propiedad de las puertas). En resumen:

Un comienzo o destino será descartado si:

- no tiene al menos dos líneas entre la etiqueta correspondiente (`<COMIENZO>`, `<DESTINO>`) y la siguiente etiqueta, y si es así la segunda línea debe tener 2 elementos (las coordenadas). Si los tiene, estos elementos siempre serán números enteros;
- las coordenadas no se corresponden con una posición válida en el mapa (en el rango de valores posibles y que además contenga una '`l`');

Una puerta será descartada si:

- no tiene tres líneas entre la etiqueta correspondiente (`<PUERTA>`) y la siguiente etiqueta, la primera con un elemento, la segunda con dos, y la tercera con 1;
- el número de puerta es negativo;
- el tercer elemento del ítem es una 'F';
- si las coordenadas no se corresponden con una posición válida en el mapa (en el rango de valores posibles y que además contenga una '`l`');

La información almacenada en el fichero se leerá automáticamente y se almacenará en estructuras de datos adecuadas para su posterior procesamiento. En concreto, para esta práctica se usará para la construcción del mapa un array bidimensional de `Casilla`.

La clase `Casilla` que hay que implementar contendrá:

- las variables de instancia que consideréis más oportunas;
- de forma que los siguientes métodos de instancia se puedan ejecutar:
  - `public Casilla(char a)`: donde el carácter indica si es libre ('l'), obstáculo ('o'), puerta ('p'), destino ('d') o comienzo ('c').
  - `public boolean setCoordenadas(int g, int n, Plano m)`: crea un objeto de tipo `Coordenadas` con los datos pasados como parámetro, siendo `g` la fila y `n` la columna, y lo almacena como su posición en la casilla. Si la casilla ya tenía unas coordenadas asignadas, o alguna de las coordenadas es errónea, no se hace nada y devuelve `false`, y si no las tiene devuelve `true`.

- `public void setNombre(String n)`: donde el String indica el nombre del comienzo o destino, en caso contrario no se hace nada;
- `public void setPuerta(int y, char p)`: donde el entero indica el número de puerta y el carácter la propiedad de la puerta, en caso de no ser una puerta no se hace nada;
- `public boolean esComienzo()`: nos devuelve `true` o `false`;
- `public boolean esDestino()`: nos devuelve `true` o `false`;
- `public boolean esObstaculo()`: nos devuelve `true` o `false`;
- `public boolean esLibre()`: nos devuelve `true` o `false`;
- `public boolean esPuerta()`: nos devuelve `true` o `false`;
- `public Coordenadas getCoordenadas()`: nos devuelve las coordenadas de la casilla;
- `public String getNombre()`: en caso de ser un comienzo o destino, nos devuelve su nombre, en caso contrario devuelve `null`;
- `public int getNumero()`: en caso de ser una puerta, nos devuelve su número, en caso contrario devuelve `-1`;
- `public char getPropiedad()`: en caso de ser una puerta, nos devuelve su propiedad, en caso contrario devuelve `'F'`;
- `public void escribeInfo()`: muestra por pantalla la información de la casilla; esta información se presentará en el siguiente orden, y separados por ':' en el caso de ser puerta, destino o comienzo:
  - nombre (si es comienzo o destino) o número (si es puerta)
  - coordenadas con el formato ``x,y''
  - propiedad, en el caso de ser una puerta

  En el caso de ser simplemente una casilla libre u obstáculo se mostrará las cadenas "libre" o "obstaculo" respectivamente.

  Por ejemplo, la salida para el destino Linz en el ejemplo del enunciado será:

  ```
  Linz:(0,3)
  ```

  para la puerta 1

  ```
  1:(1,6):P
  ```

  y para una casilla obstáculo

  ```
  obstaculo:(0,1)
  ```

- `public boolean equals(Casilla c)`: devuelve `true` si ambas casillas son iguales, `false` si no lo son.

La clase `Coordenadas` se definirá con:

- las siguientes variables de instancia:
  - `private int fila;`
  - `private int columna;`
- y los siguientes métodos de instancia:
  - `public Coordenadas(int g,int m);`
  - `public int getFila()`: devuelve la coordenada fila;
  - `public int getColumna()`: devuelve la coordenada columna;

La clase `Plano` se definirá con:

- las siguientes variables de instancia:
  - `private Casilla[][] pl;`
  - `private Coordenadas dimension;` *//dimensión de la matriz*
- y los siguientes métodos de instancia:
  - `public Plano(int i, int j)`: inicializará a dichos valores la dimensión del plano; en caso de que alguno de los enteros sea negativo, se asignará por defecto el valor 3 a ese número;
  - `public boolean setCasilla(Casilla x)`: añade una casilla al plano en las coordenadas indicadas en la casilla, si es posible, y dicha posición no contiene ya una casilla asignada. Si la operación se puede realizar devuelve `true` y en caso contrario devuelve `false`.
  - `public void leePlano(String f)`: lee todos los datos del plano (caracteres 'l' y 'o') desde un fichero (que habrá que abrir y cerrar, y cuyo nombre le habremos pasado por parámetro) y lo almacena en el array bidimensional de `Casillas`. Si leída la primera línea, no coincide con el número de columnas definido en el constructor, este valor se actualizará en la variable de instancia. A partir de dicha línea, si el resto de líneas tienen una longitud mayor, sólo se tendrán en cuenta los caracteres hasta la dimensión definida. Si tienen una longitud menor, se completarán con casillas de tipo libre ('l'). Si el número de líneas leídas de fichero es mayor que el número de filas definidos en el plano, habrá que redimensionar la estructura de una forma razonable (explicar en la documentación cómo lo hacéis). Si el número de líneas es menor, se completarán con casillas libres. Una

vez construido el plano, hay que leer el resto de información contenida en el fichero (destinos, puertas y comienzos) actualizando el plano con esta información.

Hay que tener en cuenta que el plano puede contener ya casillas con información antes de leer el plano, por lo que si esto ocurre para alguna posición de la matriz, su valor no se verá modificado por el nuevo.

Este método no propaga excepciones, por tanto para cualquier excepción que aparezca se tiene que tratar en el propio método, y dicho tratamiento será invocar el método `printStackTrace()` del objeto `Exception`.

- `public char consultaCasilla(Coordenadas x)`: devuelve el tipo de casilla almacenada en la posición dada por el objeto `x` en el plano (si es libre, obstáculo, comienzo, destino o puerta). En caso de que los valores del objeto `x` no se correspondan con una posición válida del plano o no exista definida una casilla en esa posición, devolverá `'F'`.
- `public char[] consultaVecinas(Coordenadas x)`: devuelve en un array de caracteres el tipo de casilla almacenada alrededor de la casilla que ocupa la posición dada por el objeto `x` en el plano (si es libre, obstáculo, comienzo, destino o puerta). En caso de que los valores del objeto `x` no se correspondan con una posición válida del plano o no exista definida una casilla en esa posición, devolverá `null`. El array de caracteres presentará los tipos de las casillas vecinas empezando con la casilla que se encuentra arriba de su posición y siguiente el sentido horario. En el caso de que alguna vecina sea `null` se devolverá 'F' en esa posición.

  Ejemplo, si preguntamos en el plano de la página 2 por la casilla que se encuentra en la posición (3,1), que es un comienzo, el array almacenará, en este orden: 'o','l','l','l','p','l','l','o'. En el caso del destino que ocupa la posición (6,1), el array almacenará, en este orden: 'l', 'o', 'l', 'l','l'.
- `public void muestraPlano()`: muestra por pantalla el contenido del plano, indicando sólamente si se trata de de una posición libre (l), obstáculo (o), puerta (p), comienzo (c), o destino (d); esta información se mostrará por filas, sin ninguna separación entre los símbolos. Por ejemplo, en el caso del ejemplo en el que nos estamos basando, la salida sería:

  ```
  loodllllo
  lllllplll
  oololloll
  lcllollll
  lpllodolo
  llolpodpo
  ldlloolll
  ```

  En caso de que alguna posición contenga una casilla nula mostrará 'F'.
- `public boolean equals(Plano compara)`: devuelve `true` si ambos planos son iguales, `false` si no lo son;

Hay que recordar que vosotros debéis definir las variables de instancia de la clase `Casilla` para poder leer, escribir y gestionar todo lo referente a la información que contiene. Para ello, es posible que tengáis que añadir métodos en esta clase (u otras), aparte de las que se han definido en este enunciado.

Aplicación

Supongamos que tenemos un plano, que leeremos de fichero, en el que tenemos las casillas libres, obstáculos y destinos definidos (no habrá comienzos ni puertas). Queremos encontrar una casilla que se definirá como "comienzo" y que se encuentre a la menor distancia posible de todos los destinos definidos.

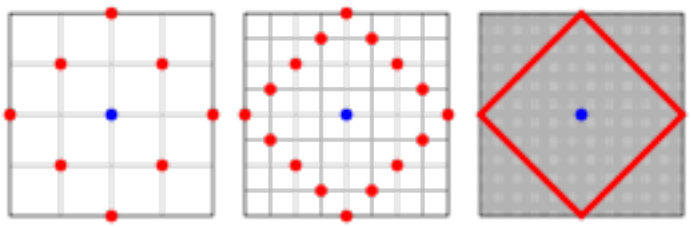
Hay que mostrar dos tipos de soluciones en dos formatos:

1. la casilla es uno de los destinos (es decir, buscamos el destino que se encuentra más cercano al resto); esto se obtiene encontrando el destino cuya suma de distancias al resto es mínima;

2. la casilla puede ser cualquiera del plano, con la única restricción que no puede ser un obstáculo. Es decir, encontrar la casilla del plano cuya suma de distancias a todos los destinos es mínima.

La distancia entre dos casillas a utilizar es la *distancia de Manhattan (o Taxicab)*, que se define como $d(c_1, c_2) = |x_1 - x_2| + |y_1 - y_2|$, donce c_1 y c_2 son casillas, y (x_1, y_1) y (x_2, y_2) las coordenadas de dichas casillas.

Geométricamente esta distancia se representa como en la siguiente figura:



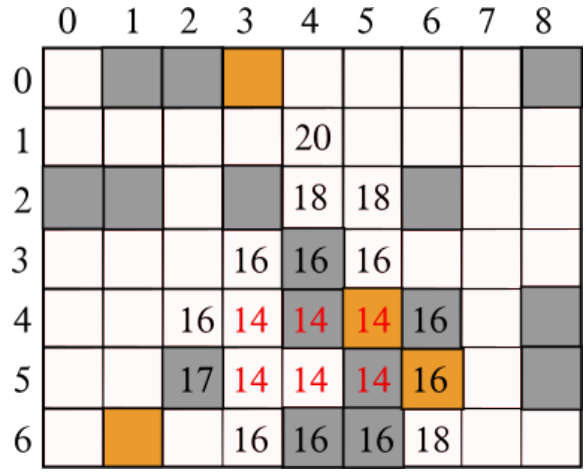
donde se puede ver que todos los puntos en rojo se encuentran a la misma distancia del punto azul.

Implementa una clase denominada `Mediaplan` que recibirá como parámetros de entradas tres argumentos, donde:

- `arg1` representa el fichero con la información del plano
- `arg2` representa las filas del plano
- `arg3` representa las columnas del plano
- `arg4` representa el formato de salida pedido, que podrá ser 'a' o 'b'

En la clase `Mediaplan` se ejecutará la aplicación que consiste en mostrar las soluciones a las dos cuestiones planteadas, una por línea. Es posible que haya más de una solución para cada uno de los casos, por lo que se mostrarán todas las posibles ordenadas por fila, y si la fila es igual, ordenadas por columna.

Si partimos de que tenemos el plano del ejemplo de la práctica, en el que no tenemos puertas, y dichas posiciones son consideradas casillas libres, la solución (entre las 4 posibles) para la primera parte es la casilla con coordenadas (4,5) con distancia 14. Para la segunda parte, mostramos a continuación la distancia de algunas casillas, entre las que están las más prometededoras, para la solución que nos piden:



La salida del formato 'a' para este ejemplo será mostrar sólo las casillas con la distancia menor:

1:14-(4,5)
2:14-(4,3)(4,5)(5,3)(5,4)

se puede observar que aunque hay 6 soluciones para la segunda parte, sólo hay que mostrar las que ocupan casillas libres o de destino (no de obstáculo). También puede ocurrir que la solución a la primera parte no tenga por qué coincidir con una de las soluciones de la segunda.

La salida del formato 'b' para este ejemplo será mostrar todas las casillas ordenadas por distancia de menor a mayor:

1:14-(4,5)
1:16-(5,6)
1:20-(6,1)
1:22-(0,3)
2:14-(4,3)(4,5)(5,3)(5,4)
2:16-(3,3)(3,5)(4,2)(5,6)(6,3)
2:18-(2,4)(2,5)(3,2)(3,6)(4,1)(5,1)(6,2)(6,6)
2:20-(1,3)(1,4)(1,5)(2,2)(3,1)(4,7)(5,7)(6,1)
2:22-(0,3)(0,4)(0,5)(1,2)(1,6)(3,7)(4,0)(5,0)(6,7)
2:24-(0,6)(1,1)(2,7)(3,0)(6,0)

2:26-(1,7)(3,8)(6,8)
2:28-(0,7)(1,0)(2,8)
2:30-(0,0)(1,8)

Si no hay solución en cualquier de los dos formatos, se muestra por pantalla el mensaja "NO HAY COMIENZO".

Documentación

Documenta en el mismo código (con un comentario antes de cada variable o método) **como mínimo** los siguientes elementos, con una breve descripción (o más extensa, dependiendo de las aclaraciones añadidas en cada método que se describe a continuación):

- las variables y métodos de instancia (o clase) añadidos por tí, justificando su necesidad e incluyendo la palabra "NEW" delante;
- las variables de instancia de la clase `Casilla`;
- la aplicación y el coste de las dos soluciones pedidas en dicha aplicación en el caso 'a', en función del número de filas (n), número de columnas (m) y número de destinos (d); indica también qué se podría hacer para mejorar computacionalmente la solución del algoritmo, en caso de que tengas claro que tu solución no es la más eficiente;
- los siguientes métodos de instancia:
  - de la clase `Plano`:
    - `public void leePlano(String f)`
    - `public boolean equals(Plano compara)`

Apéndice 1: lectura de fichero de texto

La lectura de un fichero de texto en Java, tal y como se vio en la asignatura de Programación II, se puede realizar de la siguiente manera:

- Abrir el fichero utilizando la clase FileReader y después leerlo utilizando la clase BufferedReader para leer por líneas. Por ejemplo:

```
import java.io.*;
// estas serian las variables de instancia
FileReader fichero=null;
BufferedReader lectura=null;

try{
// esto podria ir al metodo abre
 fichero=new FileReader("myfile.txt");
 lectura=new BufferedReader(fichero);
// esto podria ir al metodo lee
 String linea = lectura.readLine();
// bucle lectura hasta final de fichero
 while(linea!=null) {
  linea=lectura.readLine(); //procesar linea
 }
}catch(IOException e){
 System.err.println("Error con myfile.txt");
 System.exit(0);
}
// esto podria ir al metodo cerrar
 try{
  if (fichero!=null)
   fichero.close();
  if (lectura!=null)
   lectura.close();
 }catch(IOException ex){
   System.out.println(ex);
 }
```

Apéndice 2: Segmentación de una cadena en distintos elementos

Una vez tenemos una línea del fichero en una variable de tipo `String`, en los ejemplos la variable `linea`, hay que segmentarla para obtener de ella las distintas traducciones que necesitamos para construir un objeto de tipo **Palabra**.

La segmentación la podemos realizar utilizando el método `split` de la clase `String` de Java que ya se vio en Programación II:

- Este método devuelve un array de `Strings` a partir de uno dado usando el separador que se especifique. Por ejemplo, supongamos que estamos leyendo el diccionario y tenemos la variable `linea` de tipo `String` que contiene la cadena

  ```
  3 5
  ```

  de la cual queremos obtener por separado los diferentes elementos que contiene.

  Para ello primero debemos indicar el separador y después segmentar usando el método `split`.

  Al método `split` le podemos indicar que el separador pueden ser distintos caracteres utilizando una expresión regular, que indicamos de la siguiente manera [2]:

  ```
  //indicamos el separador de campos
   String separador = "[ ]";
  ```

  ```
  //segmentamos
   String[] elems = linea.split(separador);
  // En el caso que elems sea una línea del
  // texto, cada elemento de elems tendra
  // almacenada una palabra segmentada, es decir
     , una cadena con cada número
  ```

[2] obsérvese que esta expresión regular representa que el separador es el carácter "*" que puede ir acompañado o no (por delante y/o por detrás) de varios espacios en blanco. Esto viene representado también por el símbolo "*".

Normas generales

Entrega de la práctica:

- Lugar de entrega: servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`
- Plazo de entrega: desde el lunes 10 de octubre hasta el **viernes 14 de octubre**.
- Se debe entregar la práctica en un fichero comprimido con todos los ficheros `.java` creados y ningún directorio de la siguiente manera

  `tar cvfz practica1.tgz *.java`
- No se admitirán entregas de prácticas por otros medios que no sean a través del servidor de prácticas.
- El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud.
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada.
- Los programas deben poder ser compilados sin errores con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector inmediatamente antes de entregarla.
- Los ficheros fuente deber estar adecuadamente documentados usando comentarios en el propio código, sin utilizar en ningún caso acentos ni caracteres especiales.
- Es imprescindible que se respeten estrictamente los formatos de salida indicados ya que la corrección se realizará de forma automática.
- Al principio de cada fichero entregado (primera línea) debe aparecer el DNI y nombre del autor de la práctica (dentro de un comentario) tal como aparece en UACloud (pero sin acentos ni caracteres especiales).

  Ejemplo:
  `DNI 23433224 MUÑOZ PICÓ, ANDRÉS` $\Rightarrow$ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` $\Rightarrow$ SI

Sobre la evaluación en general:

- La práctica debe ser un trabajo original de la persona que entrega; en caso de detectarse indicios de copia de una o más entregas se suspenderá la práctica con un 0 a todos los implicados.
- La influencia de la nota de esta práctica sobre la nota final de la asignatura se detallan en las transparencias de presentación de la misma.
- La nota del apartado de documentación supone el 10 % de la nota de la práctica.

Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).
- El corrector viene en un archivo comprimido llamado `correctorP1.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP1.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica1-prueba`: dentro de este directorio están los ficheros
    - `p01.pl`: fichero de texto con el plano usado en esta prueba;
    - `p01.java`: programa en Java con un método main que realiza una serie de pruebas sobre la práctica.
    - `p01.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p01.java`.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (sólo aquellos con extensión .java) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución. Para ello ejecutar:

```
chmod +x corrige.sh
corrige.sh
```