Entregable 1
Recorrido en anchura de un laberinto

EI1022/MT1022 - Algoritmia 2023/2024

Fecha de entrega: miércoles 18 de octubre de 2023

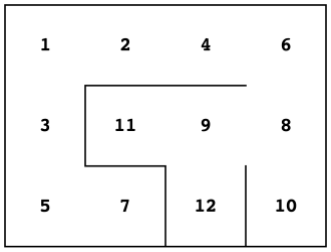
Contenido

1. El problema

Disponemos de un laberinto bien formado de R filas y C columnas. Con bien formado queremos decir que hay exactamente un camino entre cada par de celdas. Partiendo de la celda superior izquierda, $(0, 0)$, queremos averiguar la longitud del recorrido "a pie" más corto que visita todas las celdas en el orden de un recorrido en anchura. Formalmente, queremos encontrar el camino más corto tal que cuando visita una celda por primera vez ya ha visitado las anteriores según el orden de primero en anchura.

Dado que pueden existir diferentes recorridos en anchura según el orden en el que visitemos los sucesores de cada vértice, vamos a fijar un orden concreto. Desde un vértice (r, c), el orden en el que visitaremos sus sucesores, si existen, será $(r-1,\ c)$, $(r,\ c-1)$, $(r,\ c+1)$ y $(r+1,\ c)$, es decir, arriba, izquierda, derecha y abajo.

Veamos el ejemplo de este laberinto (instancia `lab_01_000012.i`):



Los números en el laberinto indican el orden del recorrido en anchura que hemos fijado. Por comodidad, vamos a utilizar estos números para identificar las celdas en lugar de sus coordenadas.

Partimos de la celda con el número 1 y ponemos nuestro contador de pasos a cero. Desde la celda 1 vamos a la 2 en un paso, desde la 2 vamos a la 3 en dos pasos (tenemos que pasar por la 1), desde la 3 vamos a la 4 en tres pasos (tenemos que pasar por la 1 y la 2), etc. Cuando lleguemos a la celda 12 habremos andado $1 + 2 + 3 + 4 + 5 + 6 + 7 + 1 + 2 + 3 + 2 = 36$ pasos, que es la solución buscada.

Podéis ver un video de cómo se recorre el camino en el laberinto anterior en este enlace. Tenéis otro video similar para la instancia `lab_02_000048.i` en este otro enlace.

1.1. Implementación

Tenéis que implementar un programa de línea de órdenes, `e1.py`, que reciba, por la entrada estándar, un laberinto en el formato que se especifica en el apartado 1.2. Como resultado de su ejecución, el programa deberá mostrar sus resultados por la salida estándar en el formato que se detalla en el apartado 1.3.

Para conseguir la máxima nota, el coste temporal asintótico del programa deberá ser lineal con el tamaño del laberinto (grafo), entendido como el número total de celdas (vértices).

1.2. Formato de la entrada

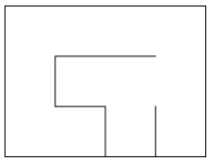
Una instancia de laberinto consiste en varias líneas de texto, la primera línea contiene dos enteros separados por un blanco, que representan, respectivamente, el número de filas, R, y de columnas, C, del laberinto. Las siguientes R líneas representan el laberinto. Concretamente, cada línea representa una fila del laberinto mediante una cadena de longitud C formada con los números del 0 al 3, con este significado:

- 0: la celda tiene pared a su derecha y abajo.
- 1: la celda se comunica con la de su derecha y tiene pared abajo.
- 2: la celda se comunica con la de abajo y tiene pared a su derecha.
- 3: la celda se comunica con la de su derecha y la de abajo.

Como podéis ver, no aparecen referencias ni al vecino izquierdo ni al superior. Dado que el grafo que representa el laberinto es no dirigido, podemos describirlo perfectamente de este modo.

Veamos, por ejemplo, la instancia `lab_01_000012.i`. A la izquierda se muestran las cuatro líneas que contiene el fichero y a la derecha el laberinto que representa:

```
3 4
3112
2132
1000
```



1.3. Formato de la salida

El programa deberá mostrar, por la salida estándar, el número de pasos del recorrido. Por ejemplo, para la instancia `lab_01_000012.i` deberá mostrar:

```
36
```

2. Ficheros de partida

En el aula virtual disponéis de una carpeta con el siguiente contenido:

- `public_test.zip`: un archivo comprimido que contiene la carpeta `public_test` con un conjunto de instancias de prueba y sus soluciones.
- `e1.py`: el programa que debéis de utilizar como punto de partida y que hay que entregar.
- `e1_test.py`: programa que importa `e1.py` y lo utiliza para validar tanto la solución obtenida como el tiempo que la función `process` ha necesitado para obtenerla.
- `e1_lab_viewer.py`: un programa que te permite visualizar los laberintos de las instancias de prueba.

Los siguientes apartados explican con detalle estos contenidos.

2.1. Ficheros de prueba (public_test.zip)

Al descomprimir el fichero `public_test.zip` obtendréis la carpeta `public_test`, que contiene diez instancias de prueba y sus soluciones, los ficheros `lab_*.i` y `lab_*.o`, respectivamente.

Las diez instancias podemos dividirlas en tres bloques según su tamaño:

- Instancias de la 1 a la 4: instancias pequeñas.
- Instancias de la 5 a la 8: instancias medianas.
- Instancias 9 y 10: instancias grandes.

Para ayudaros en la depuración, las instancias 1 y 2 tienen un fichero adicional (con extensión ".v"), que contine los vértices del recorrido paso a paso.

2.2. Fichero principal (e1.py)

Contiene la estructura del programa. Debéis modificarlo teniendo en cuenta lo siguiente:

- Podéis añadir funciones o clases adicionales, pero **no debéis modificar nada del programa principal**. Modificar el programa principal supondrá un cero en la calificación del entregable. Tampoco podéis modificar la firma (el tipo) de las tres funciones que se utilizan en el programa principal.
- Podéis importar módulos de la biblioteca estándar de Python, pero la única biblioteca externa permitida es `algoritmia`. Por ejemplo, NumPy es una biblioteca externa y, por lo tanto, no está permitida.

Veamos la estructura de `e1.py` y las funciones que debéis implementar.

Por comodidad, el programa define los tipos `Vertex` y `Edge`:

```
Vertex = tuple[int, int]
Edge   = tuple[Vertex, Vertex]
```

El programa principal utiliza la estructura de tres funciones vista en clase:

```
graph0, rows0, cols0 = read_data(sys.stdin)
steps0 = process(graph0, rows0, cols0)
show_results(steps0)
```

La función **show_results** es trivial, ya está implementada y no debéis modificarla. Debéis implementar las funciones **`read_data`** y **`process`**:

- **`read_data`**`(f: TextIO) -> tuple[UndirectedGraph[Vertex], int, int]`

  Entrada: El fichero con la instancia (ojo, `f` no es un nombre de fichero, es un fichero).

  Salida: Una tupla de tres elementos, el primero es el grafo no dirigido que representa el laberinto, el segundo es el número de filas del laberinto y el tercero el número de columnas.

- **`process`**`(lab: UndirectedGraph[Vertex], rows: int, cols: int) -> int`

  Entrada: Tres parámetros, los mismos que forman la tupla de salida de la función `read_data`.

  Salida: Un entero indicando el número de pasos del recorrido.

Cómo ejecutar `e1.py` desde la línea de órdenes:

1. Abrid un terminal[1] y utilizad el comando `cd` para ir al directorio donde está el fichero `e1.py` y la carpeta `public_test`.

2. Si no la tenéis instalada ya, instalad la biblioteca `algoritmia`. Se explicó cómo hacerlo en la segunda sesión de prácticas (la información está disponibles en el aula virtual).

3. Ya podéis lanzar vuestro programa.

   En Linux y MacOS:

   Desde el terminal, ejecutad el comando:

   ```
   python3.10 e1.py < public_test/lab_01_000012.i
   ```

   que debería mostrar 36 como resultado.

   En Windows:

   Desde el terminal CMD (no `desde PowerShell`), ejecutad el comando:

   ```
   python e1.py < public_test\lab_01_000012.i
   ```

   que debería mostrar 36 como resultado.

2.3. El validador de soluciones (`e1_test.py`)

Este programa importa el fichero `e1.py` por lo que ambos deben estar en el mismo directorio.

Abrid un terminal (en Windows que sea CMD) e id al directorio donde están `e1_test.py`, `e1.py` y la carpeta `public_test`.

Ejecutad la siguiente orden, según el sistema operativo:

- En Linux y MacOS:

  ```
  python3.10 -O e1_test.py public_test/lab_01_000012.i public_test/lab_01_000012.o
  ```

- En Windows (desde un terminal CMD):

  ```
  python -O e1_test.py public_test\lab_01_000012.i public_test\lab_01_000012.o
  ```

Las ordenes anteriores utilizan la opción -O de Python para ejecutar vuestro programa con las optimizaciones activadas.

La ejecución de vuestro programa producirá una de estas cinco posibles salidas:

- SOLUTION OK - TIME OK (<num> sec)
- SOLUTION OK - TIME ERROR (<num> sec)
- SOLUTION ERROR - TIME OK (<num> sec)
- SOLUTION ERROR - TIME ERROR (<num> sec)
- El programa no termina en un plazo razonable y tenéis que pulsar `Ctrl-C` para terminarlo.

[1] Windows tiene dos terminales, el antiguo (CMD) y el nuevo (PowerShell). En la asignatura utilizaremos el antiguo. PyCharm utiliza PowerShell por defecto, pero se puede cambiar por CMD en las preferencias.

2.4. El visor de laberintos (`e1_lab_viewer.py`)

Este programa os permite visualizar los laberintos (los ficheros de instancia `lab*.i`).

Para poder leer el laberinto, este programa importa la función `read_data()` de vuestro fichero `e1.py`, por lo este programa no funcionará hasta que la implementéis.

Veamos cómo se utiliza. Abrid un terminal e id al directorio donde están `e1_lab_viewer.py`, `e1.py` y la carpeta `public_test`. Ejecutad la siguiente orden para lanzar el visor:

- En Linux y MacOS:

  ```
  python3.10 e1_lab_viewer.py < public_test/lab_01_000012.i
  ```

- En Windows (en CMD):

  ```
  python e1_lab_viewer.py < public_test\lab_01_000012.i
  ```

Se abrirá una ventana con el laberinto. Podéis utilizar las teclas “G” y “L” para ocultar/mostrar el grafo subyacente y el laberinto que representa. Pulsad la tecla `Escape` o `Return` para cerrar la ventana y terminar el programa.

3. Entrega en el aula virtual

La entrega consistirá en **subir dos ficheros** a la tarea correspondiente del aula virtual, **solo debe subirlos uno de los miembros del grupo**. Estos son los dos ficheros:

- **`e1.py`**: El fichero con el código del entregable.
- **`alxxxxxx_alyyyyyy.txt`**: Un fichero de texto que deberá contener el nombre, el número de DNI y el `al######` de cada miembro del grupo (el formato del contenido es libre). Evidentemente, en el nombre del fichero tenéis que reemplazar alxxxxxx y alyyyyyy por los correspondientes a los dos miembros del grupo.

  Si el grupo es unipersonal, el fichero de texto deberá llamarse **`alxxxxxx.txt`**, reemplazando `alxxxxxx` por el que corresponda.

Vuestra entrega debe cumplir estas restricciones (cada restricción no cumplida quita un punto del entregable):

- Los nombres de los dos ficheros deben utilizar únicamente minúsculas.
- El separador utilizado en el nombre de fichero `alxxxxxx_alyyyyyy.txt` es el guion bajo (“_”), no utilicéis un menos (“-”) ni ningún otro carácter similar.
- Debéis poner correctamente las extensiones de los dos archivos (“`.py`” y “`.txt`”). **Si utilizáis Windows es posible que tengáis las extensiones de archivo ocultas (es la configuración por defecto de Windows) por lo que es posible que enviéis ficheros con doble extensión, evitadlo: configurad vuestro Windows para que muestre las extensiones de los archivos.**
- No subáis ningún fichero ni directorio adicional.

4. Calificación del entregable

El entregable se calificará utilizando unas pruebas privadas que se publicarán junto con las calificaciones.

Las pruebas privadas consistirán en diez instancias de tamaño similar a los de las pruebas públicas.

El ordenador con el que se medirán los tiempos de ejecución será `lynx.uji.es`. Un ordenador al que todos tenéis acceso y en el que podéis ejecutar el programa `e1_test.py` que os proporcionamos.

Para considerar una instancia superada, vuestro método `process` deberá tardar **menos de un segundo** en terminar. Con los laberintos pequeños es posible conseguirlo incluso con un algoritmo muy ineficiente, con los medianos se necesita un algoritmo más eficiente. Por último, con los grandes necesitaréis un algoritmo con coste lineal con el número de celdas.

Por lo tanto, el programa puede estar mal de dos formas diferentes:

- Tiene uno o más errores y funciona mal con algunas instancias (o con todas).
- No tiene errores, pero tarda más de un segundo en obtener la solución.

Ambos problemas pueden detectarse utilizando las pruebas públicas, aunque superar las pruebas públicas no garantiza superar las privadas, sobre todo si la implementación se ha "ajustado" específicamente para superar las públicas.

4.1. Errores graves en el entregable

Obtendréis directamente una puntuación de cero en el entregable si modificáis el programa principal de `e1.py` o los tipos de cualquiera de las funciones que utiliza.

Se os penalizará también con un cero si vuestro programa no lee las instancias desde la entrada estándar, tal y como se indica en el apartado 1.2.

También se penalizará que la salida no respete el formato que se especifica en el apartado 1.3. Una salida errónea puede tener dos causas:

- Una implementación que no respeta el formato especificado. La penalización consistirá en quitar **dos puntos** a la nota del entregable.
- Algún `print` olvidado en el código y que se ejecuta durante las pruebas. La penalización consistirá en quitar **un punto** a la nota del entregable.

Po último, también se penaliza con un punto cada restricción incumplida en la entrega al aula virtual (ver el apartado 3).

Revisadlo con detenimiento antes de entregar (todos los miembros del grupo).

4.2. Penalización por copia

En caso de detectarse una copia entre grupos, se aplicará la normativa de la universidad: la calificación de la evaluación continua (60 % de la nota final) será de cero en la primera convocatoria para todos los miembros de los grupos involucrados.