EI1022, Algoritmia

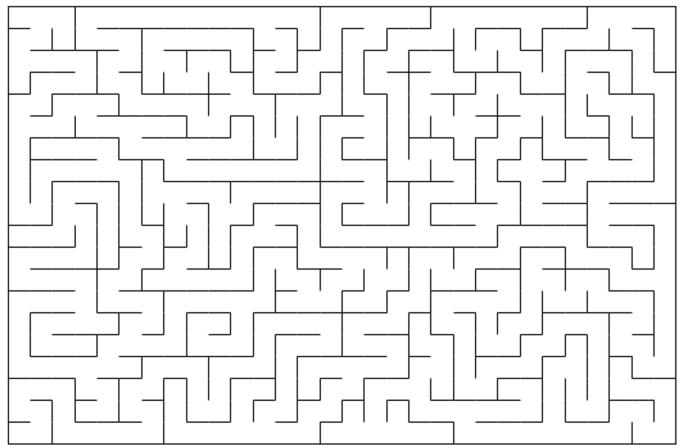
Práctica 2: Construcción de laberintos

Introducción

- En esta práctica vamos a ver cómo generar laberintos
- Escribiremos dos programas:
  - `labyrinth.py`: que generará un laberinto aleatorio a partir de un número de filas y columnas
  - `verLaberinto.py`: que visualizará un laberinto por pantalla

Construcción de laberintos

- Utilizando un MFSet podemos construir laberintos similares a este:



- Estos laberintos tienen la propiedad de que entre cualquier par de celdas hay un único camino.

Construcción de laberintos (2)

Un laberinto se puede representar fácilmente mediante un grafo no dirigido:

- Un vértice por cada celda:
  - Tuplas $(fila, columna)$ con las coordenadas de la celda.
  - Ejemplo: $(2, 3)$ representa la celda de la fila 2, columna 3.
- Una arista por cada par de celdas vecinas comunicadas
  - Una arista es simplemente una tupla de dos vértices.
  - Ejemplo: la arista $((2, 3), (2, 4))$ dice que se puede pasar desde la celda $(2, 3)$ a la $(2, 4)$.

Implementación

- Seguiremos el esquema
- Los tipos:

```
type Data = tuple[int, int]
type Result = UndirectedGraph
```

- La función `read_data`:

```
def read_data(f: TextIO) -> Data:
    rows = int(f.readline())
    cols = int(f.readline())
    return rows, cols
```

Implementación (2)

- Nuestro `process` creará grafo del laberinto a partir del número de filas y columnas
- Su cabecera es:

```
def process(data: Data) -> Result:
```

Implementación (3)

- `show_result` simplemente escribe el grafo por pantalla:

```
def show_result(result: Result):
    print(result)
```

- El programa principal sigue el esquema:

```
if __name__ == "__main__":
    data0 = read_data(sys.stdin)
    result0 = process(data0)
    show_result(result0)
```

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).
- Paso 2: Crea un *MFSet* vacío usando la función `MergeFindSet()`, del módulo `algoritmia.datastructures.mergefindsets`. Añádele los vértices de `vertices` usando `mfs.add(·)`.

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).
- Paso 2: Crea un *MFSet* vacío usando la función `MergeFindSet()`, del módulo `algoritmia.datastructures.mergefindsets`. Añádele los vértices de `vertices` usando `mfs.add(·)`.
- Paso 3: Crea una lista, `edges`, con todos los pares de vértices vecinos y barájala. Usa la función `shuffle` del módulo `random`.

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).
- Paso 2: Crea un *MFSet* vacío usando la función `MergeFindSet()`, del módulo `algoritmia.datastructures.mergefindsets`. Añádele los vértices de `vertices` usando `mfs.add(·)`.
- Paso 3: Crea una lista, `edges`, con todos los pares de vértices vecinos y barájala. Usa la función `shuffle` del módulo `random`.
- Paso 4: Crea una lista vacía, `corridors`. Aquí pondremos las aristas (pasillos) que tendrá al final nuestro grafo (laberinto).

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).
- Paso 2: Crea un *MFSet* vacío usando la función `MergeFindSet()`, del módulo `algoritmia.datastructures.mergefindsets`. Añádele los vértices de `vertices` usando `mfs.add(·)`.
- Paso 3: Crea una lista, `edges`, con todos los pares de vértices vecinos y barájala. Usa la función `shuffle` del módulo `random`.
- Paso 4: Crea una lista vacía, `corridors`. Aquí pondremos las aristas (pasillos) que tendrá al final nuestro grafo (laberinto).
- Paso 5: Recorre la lista `edges` y, para cada arista `(u,v)`, encuentra la clase a la que pertenece cada uno de los dos vértices usando `msf.find(·)`. Si son diferentes, fusiónalas en la misma clase con `mfs.merge(u, v)` y añade la arista `(u,v)` a la lista `corridors`.

Implementación de `process`

- Paso 1: Crea una lista, `vertices`, con los vértices del grafo (las celdas del laberinto).
- Paso 2: Crea un *MFSet* vacío usando la función `MergeFindSet()`, del módulo `algoritmia.datastructures.mergefindsets`. Añádele los vértices de `vertices` usando `mfs.add(·)`.
- Paso 3: Crea una lista, `edges`, con todos los pares de vértices vecinos y barájala. Usa la función `shuffle` del módulo `random`.
- Paso 4: Crea una lista vacía, `corridors`. Aquí pondremos las aristas (pasillos) que tendrá al final nuestro grafo (laberinto).
- Paso 5: Recorre la lista `edges` y, para cada arista `(u,v)`, encuentra la clase a la que pertenece cada uno de los dos vértices usando `msf.find(·)`. Si son diferentes, fusiónalas en la misma clase con `mfs.merge(u, v)` y añade la arista `(u,v)` a la lista `corridors`.
- Paso 6: Construye el grafo y devuélvelo:

```
return UndirectedGraph(E = corridors)
```

Visualizar laberintos

- En `verLaberinto.py` usamos la clase `LabyrinthViewer` de la biblioteca `algoritmia` para visualizar laberintos:

```
from algoritmia.viewers.labyrinth_viewer import LabyrinthViewer
from labyrinth import process

if __name__ == "__main__":
    rows = int(input("Filas: "))
    cols = int(input("Columnas: "))
    data = (rows, cols)
    labyrinth = process(data)
    lv = LabyrinthViewer(labyrinth,
                         canvas_width=10 * cols,
                         canvas_height=10 * rows)
    lv.run()
```

Visualizar laberintos (2)

En la ventana de visualización se pueden usar las siguientes teclas:

- La tecla `l` muestra u oculta los muros del laberinto
- La tecla `g` muestra u oculta el grafo correspondiente al laberinto
- La barra de espacio muestra solo los muros o el grafo
- Con `esc` o `return` se sale del programa