Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2018-2

27 de septiembre de 2018
Evaluada

Actividad 07

Estructuras de Datos: *grafos*

Introducción

El DCC[1] ha comenzado un nuevo e innovador proyecto de viviendas, *DCCity*, pero el malvado doctor *Evil Kawas*[2], extremo partidario de las construcciones tradicionales, intenta detenerlos. Para ello, con la ayuda del jefe ~~de docencia~~ de la mafia *Tini Tamburini*, escondió bombas en distintas casas de la ciudad. Será el trabajo del famoso *James Castro*, el héroe favorito de todos los niños de *IIC2233* desactivar dichas bombas. Sin embargo, también necesitará la ayuda del miembro más reciente del equipo, *Philip Rivers*, para encontrarla.

Mientras James conduce y Philip crea un código capaz de desactivar la bomba, será tu trabajo desarrollar un software capaz de guiar a nuestros héroes. Para lo anterior, deberás crear un mapa de la ciudad a partir de los archivos que te entregará la constructora y crear el software *D.N.W.*[3] capaz de entregar rutas a partir de los mapas creados.

1. La Ciudad

A diferencia de la mayoría de las ciudades, cada casa de *DCCity* tiene su propio terreno aislado y existen calles que unen estos terrenos. Al igual que en las ciudades normales, dichas calles pueden ser de uno o dos sentidos. Esto implica que existen ciertos caminos por los que puedes ir pero no devolverte, y viceversa.

Dado que el proyecto está en sus etapas más tempranas, no existe un mapa de la ciudad, por lo que es tu deber crearlo a partir de los archivos `facil.txt`, `medio.txt`, `dificil.txt` y `kratos.txt` que modelan ciudades con distinta distribución y cantidad de casas. Dichos archivos poseen líneas con el siguiente formato:

```
D: B,C,E,H
```

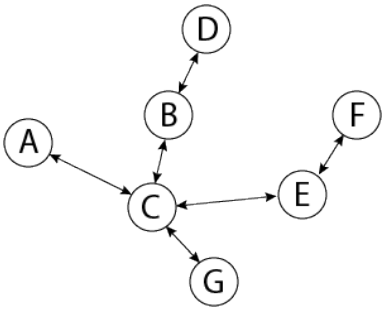
En estos archivos, el *string* que antecede al signo `:` representa un terreno (terreno de origen), y el *string* que sucede al mismo signo, representan los terrenos con los que este está conectado (terrenos destino), es decir,

```
A: B,G,D
```

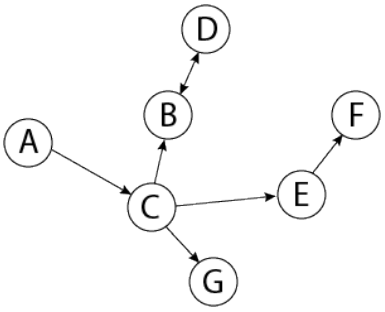
significa que existen los caminos `A -> B`, `A -> G` y `A -> D` y, por supuesto, que existen los terrenos `A, B, D` y `G`. En cada línea, el archivo tendrá solamente un terreno de origen y un número indefinido

[1] *Departamento de Ciencias de la Construcción.*
[2] Cualquier parecido con la realidad es mera coincidencia.
[3] *Definitely Not Waze.*

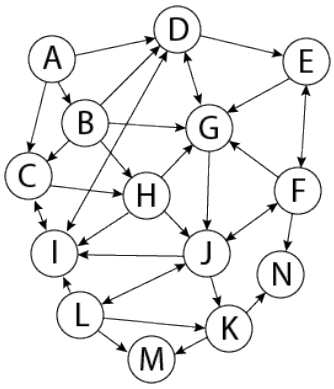
de terrenos destino separados por comas (`,`). Claramente, el mapa de la ciudad será un **grafo dirigido** y para ayudarte a hacer pruebas, a continuación tenemos tres imágenes correspondientes a los grafos de `facil.txt`, `medio.txt` y `dificil.txt`.



`facil.txt`



`medio.txt`



`dificil.txt`

2. D.N.W.

Además de guardar los distintos mapas, el programa DNW deberá ser capaz de agregar nuevas calles entre los terrenos, eliminar calles existentes (no quieres que tu programa no tenga arreglo), imprimir de manera **legible** el grafo de cada mapa, señalar si un cierto camino indicado es posible de realizar en *DCCity* y encontrar un camino entre distintas bombas. A su vez, en caso de existir el camino, deberá entregar una ruta correcta para llegar desde un terreno a otro, en cualquier ciudad (no solamente aquellas modeladas con los archivos `facil.txt`, `medio.txt`, `dificil.txt` y `kratos.txt`).

Para poder llevar a cabo tu preciado software, deberás implementar una clase llamada `Terreno` y otra clase llamada `Ciudad`. La primera debe implementarse como nodo, mientras que la segunda deberá incluir los siguientes métodos **obligatorios** para lograr un correcto funcionamiento.

- `def __init__(self, path):`

  Recibe como parámetro el *path* hacia algún archivo con el formato comentado anteriormente, y crea el grafo a partir de este.

- `def agregar_calle(self, origen, destino):`

  Recibe como parámetro el nombre del terreno de origen y el nombre del terreno de llegada, creando una conexión (calle) de un sentido desde el terreno de origen, hacia el terreno de destino. En caso de que alguno de los terrenos no se encuentre previamente en el grafo, debe ser creado antes de agregar la conexión. Si la calle ya existe, no se realiza ninguna acción.

- `def eliminar_calle(self, origen, destino)`

  Recibe como parámetro el terreno de origen y el terreno de destino. Se debe eliminar la calle que los une (que tiene sentido desde el origen hacia el destino), pero el resto de las calles debe perdurar. Además, debe retornar la tupla `(origen, destino)` si la operación fue realizada y una tupla vacía si no es posible realizarla (si no existe alguno de los nodos, si no existe la conexión entre origen y destino, etcétera).

- `def terrenos(self):`

`terrenos` debe ser un `property` que retorne el conjunto con los nombres de los terrenos (nodos) del mapa (grafo); cabe señalar que el conjunto de terrenos debe actualizarse acorde a los cambios realizados a través de `agregar_calle`.

- `def calles(self):`

  `calles` debe ser un `property` que retorne el conjunto de calles (arista) del mapa (grafo); la manera de representar las calles **debe ser** con una tupla `(origen, destino)`, en donde `origen` y `destino` es el nombre del terreno de origen y el de destino, respectivamente. Es importante notar que esta property deberá retornar un conjunto de tuplas. Al igual que para `terrenos` los cambios realizados al mapa, deben verse reflejados en el conjunto de calles.

- `def verificar_ruta(self, ruta):`

  Recibe como parámetro una ruta entre terrenos (nodos), en forma de lista, y retorna un booleano indicando si dicho camino es válido dado el mapa de la ciudad. Dicha lista entrega el camino en orden. Así, a modo de ejemplo, la lista `['A', 'B', 'C']` representa el camino que va de `A` a `B`, y luego a `C`. Cabe señalar que los caminos vacíos serán siempre válidos, y los caminos con un solo nodo lo serán si este nodo pertenece al grafo.

- `def entregar_ruta(self, origen, destino):`

  Recibe como parámetros los terrenos de origen y destino, y retorna una lista representando algún camino entre tales terrenos. El formato de la lista es el mismo que el indicado en `verificar_ruta`. Si el origen es igual al destino, se debe entregar una lista que sólo contiene a tal terreno. Para cualquier otro caso (no es posible llegar de un nodo a otro, alguno de los nodos no está en el grafo, etc), debe retornar una **lista vacía**.

  ¡Bonus! Si logras que tu método `entregar_ruta` retorne la ruta más corta entre ambos terrenos, puedes obtener una bonificación en tu nota (ver detalles en la sección Bonus).

- `def ruta_entre_bombas(self, origen, *destinos):`

  Recibe como parámetros un terreno de origen y un número arbitrario[4] de terrenos donde hay bombas. Retorna una lista que indica alguno de los caminos que comienzan en el nodo de origen, terminan en el último nodo destino y pasan, en orden, por todo el resto de destinos entregados. El formato de la lista **debe** ser de la misma forma que en `verificar_ruta`; en caso de no existir ningún camino que cumpla las condiciones, se debe retornar una **lista vacía**.

  Cabe señalar que el camino puede pasar más de una vez por un mismo nodo incluyendo al nodo final, es decir, para el origen `A` y los destinos `H`, `D`, `K`, el camino `['A', 'C', 'D', 'H', 'K', 'D', 'K']` será válido, pues comienza en el nodo origen, pasa por todos los nodos destino, en orden y finaliza en el último nodo destino.

  ¡Bonus! Si logras que tu método `ruta_entre_bombas` retorne la ruta más corta entre los dos nodos, puedes obtener una bonificación en tu nota (ver detalles en la sección Bonus).

Notas

- **Es obligatorio utilizar los nombres indicados en esta actividad tanto para la clase `Terreno` y `Ciudad` como para sus métodos; no cumplir con esto significará un descuento de hasta 5 décimas en tu nota.**

[4] Asuman que **siempre** se pasa al menos un destino. No es necesario que manejen para el caso en que no se pasen destinos como argumento.

- Es recomendable resolver los métodos en orden. Así, podrás utilizarlos cuando sea conveniente.
- Debes utilizar las estructuras de datos aprendidas últimamente (y las que ya conocías) de manera efectiva. Un uso poco adecuado de estas puede significar un descuento de tu puntaje.
- Puedes utilizar funciones auxiliares en caso de ser necesario.
- Tu solución puede ser tanto recursiva como iterativa.

Bonus (1 punto)

Si lograste hacer que `entregar_ruta` y/o `ruta_entre_bombas` entreguen el camino más corto para los nodos pedidos, debes definir los siguientes dos métodos en tu código:

- `def ruta_corta(self, origen, destino):`
- `def ruta_corta_entre_bombas(self, origen, *destinos):`

Ambos métodos, solamente deben hacer `return` de `entregar_ruta` y `ruta_entre_bombas` respectivamente, pues los métodos originales deben, automáticamente encontrar el camino más corto; el implementarlos es simplemente una manera de avisar que lo lograste completar. Para ambos métodos, si existen varios caminos con el menor largo posible, basta con retornar uno de estos.

Requerimientos

- (0,50 pts) Clase `Terreno`
  - (0,50 pts) Implementar clase `Terreno`
- (5,50 pts) Clase `Ciudad`
  - (0,50 pts) Implementar `__init__` con lectura de datos.
  - (0,50 pts) Implementar `agregar_calle`.
  - (0,50 pts) Implementar `eliminar_calle`.
  - (0,50 pts) Implementar `terrenos`.
  - (0,50 pts) Implementar `calles`.
  - (1,00 pt) Implementar `verificar_ruta`.
  - (1,00 pt) Implementar `entregar_ruta`.
  - (0,50 pts) Implementar `ruta_entre_bombas`.
  - (0,50 pts) Poblar el sistema con los grafos correspondientes a los 4 archivos entregados, buscando y entregando al menos una ruta por cada grafo.
- (0,5 pts) **Bonus:** Implementar `ruta_corta`.
- (0,5 pts) **Bonus:** Implementar `ruta_corta_entre_bombas`.

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AC07/`
- **Hora del último *push* válido:** 16:40