Implementación de la clase `Graph` en `C++`*

*Junto a algunos algoritmos conocidos en la teoría de grafos no dirigidos.

Víctor Samuel Pérez Díaz
Matemáticas Aplicadas y Ciencias de la Computación
Universidad del Rosario
Bogotá D.C., Colombia
victor.perez@urosario.edu.co

Juan Esteban Murcia Nieto
Matemáticas Aplicadas y Ciencias de la Computación
Universidad del Rosario
Bogotá D.C., Colombia
juane.murcia@urosario.edu.co

***Index Terms*—grafo, algoritmo, nodo, arista, matriz**

Repositorio: https://github.com/JEMN2001/Graphs

I. Introducción

Este documento busca exponer una implementación de la clase `Graph` en el lenguaje de programación `C++`. Junto a esta, se realizaron varios algoritmos conocidos en la teoría de grafos no dirigidos. `C++`.

II. Resumen

Este proyecto se realizó con los ánimos de tener una representación de grafos no dirigidos. Junto a esta, también se realizaron varios algoritmos clásicos enseñados en nuestro curso de *Teoría de Grafos*, junto a algunos que investigamos por nuestra cuenta.

La clase Graph se basa en la matriz de adyacencia de un grafo no dirigido. Todos los algoritmos fueron construidos en base a esta.

Por tanto, la clase `Graph` cuenta con tan solo un parámetro `matrix`, en el cual se almacena la matriz de adyacencia.

Los métodos desarrollados fueron:

- `Graph` *y sus derivados.*
- `uncolored_vertex`
- `colored_vertex`
- `colored_neighbor`
- `uv_path`
- `grade`
- `empty`
- `order`
- `size`
- `max_grade`
- `min_grade`
- `remove_node`
- `remove_edge`
- `add_node`
- `add_edge`
- `clear`
- `print_matrix`
- `find_path`
- `color_graph`
- `breadth_search`
- `depth_search`
- `shortest_uv_path`

Cada uno de los anteriores será explicado en la sección **Funcionalidad**.

Identify applicable funding agency here. If none, delete this.

III. Funcionalidad

La clase `Graph` está basada en la matriz de adyacencia de un grafo no dirigido, esta siendo su único parámetro.

El primer acercamiento a la idea de la clase fue crear dos estructuras `nodos (V)` y `aristas (E)`, la construcción resulta ser conflictiva al hacer uso de punteros, los cuales interactuan con la memoria del computador. Dado esto, encontramos que la implementación a través de la matriz resultaba mucho más práctica.

Se desarrollaron varias versiones del constructor `Graph`:

- `Graph()`: Constructor vacío, un grafo sin nodos ni aristas.
- `Graph(Graph)`: Constructor copia.
- `Graph(A_matrix)`: Construye un grafo con una matriz de adyacencia específica.
- `Graph(filename)`: Construye un grafo con una matriz de adyacencia específica contenida en un archivo `.txt`.

También, se realizó el método destructor:

- $\sim$`Graph()`: Destructor de la clase.

En la **sección privada** de la clase, se definió el parámetro:

- `matrix`: Matriz de adyacencia del grafo.

Y además, se definieron los siguientes métodos:

- `uncolored_vertex`: Función que determina si un vértice no tiene color dada una coloración.
- `colored_neighbor`: Función que determina si un vértice tiene un vecino con un color dado.
- `uv_path`: Función que determina si hay un $uv-$camino en el grafo.

Ahora, en la **sección pública**, se definieron los siguientes métodos (junto a los constructores y destructor de la clase, definidos arriba):

- `grade`: Función que calcula el grado de un vértice.
- `empty`: Función que determina si un grafo no tiene vértices.

- `order`: Función que nos da el número de vértices del grafo.
- `size`: Función que nos da el número de aristas del grafo.
- `max_grade`: Función que nos da el máximo grado del grafo.
- `min_grade`: Función que nos dale el mínimo grado del grafo.
- `remove_node`: Función que remueve un vértice específico y todas sus aristas adyacentes.
- `remove_edge`: Función que remueve una arista específica del grafo.
- `add_node`: Función que inserta un vértice nuevo al grafo.
- `add_edge`: Función que inserta una nueva arista al grafo.
- `clear`: Limpia el grafo.
- `print_matrix`: Función que muestra la matriz de adyacencia del grafo.
- `find_path`: Función que determina si hay un camino euleriano en el grafo y en caso afirmativo, lo encuentra.
- `color_graph`: Función que da un coloreado apropiado a los vértices.
- `breadth_search`: Función que retorna un árbol de expansión del grafo usando el algoritmo de búsqueda a lo ancho.
- `depth_search`: Función que retorna un árbol de expansión del grafo usando el algoritmo de búsqueda a profundidad.
- `shortest_uv_path`: Función que usa el algoritmo de Dijkstra para encontrar el menor $u, v-$camino en el grafo.

IV. Descripción

En esta sección, se describirán los algoritmos que consideramos más relevantes, además, dados los análisis hechos a algunos de ellos, indicaremos su complejidad si es el caso:

- `find_path`: Lo primero que hace el algoritmo es contar la cantidad de vértices que tengan grado impar, si estos son mayores a 2, retorna una lista vacia, de lo contrario toma como inicio uno de los vértices de grado impar (cualquier vértice si no hay vértices de grado impar) recorre una arista hacia otro vértice, elimina la arista que recorrió y repite el proceso hasta que llegue a un vértice sin más aristas, al llegar añadirá ese vértice al camino y se devolverá al anterior, repitiendo el proceso hasta que no se pueda devolver más.
  Complejidad: $\mathcal{O}(N + E)$
- `color_graph`: Inicializa todos los nodos con color 0 (sin color) y mientras haya un vértice sin color, recorrera todos los vértices, si el vértice actual no tiene color y ninguno de sus vecinos tiene ese color, le asignara ese color a ese vértice, si acaba la iteración y siguen habiendo vértices sin color, cambia el color y repite el proceso.
  Complejidad: $\mathcal{O}(N^3)$
- `breadth_search`: Crea un nuevo grafo con la misma cantidad de vértices que el grafo de entrada, luego recorrerá los vértices de ese grafo, revisará la conexión del vértice actual con el resto de vértices, y agregará una arista entre un par de vértices en el grafo de salida, si esa arista está en el grafo de entrada y no existe ya un camino entre esos vértices en el grafo de salida.

- `depth_search`: Crea un nuevo grafo con la misma cantidad de vértices que el grafo de entrada, luego recorrerá todos los vertices de ese grafo, si encuentra una arista en el grafo de entrada que no este en el grafo de salida y no haya un camino entre esos dos vértices en el grafo de salida, agregará la arista y avanzará al siguiente vértice.

- `shortest_uv_path`: Lo primero que hace es el algoritmo es determinar si ambos vértices pertenecen al grafo, si esto no ocurre, soltara un mensaje de error y retornará una lista vacía. De lo contrario inicializa una lista de valores con 0 en el vértice de inicio e infinito en el resto de vértices, un conjunto con los vertices que puedo chequear y una lista con el camino que se debe tomar inicializada con infinito. Mientras el vértice de llegada siga en el conjunto, selecciona el vértice con menor valor en la lista de valores, eliminará ese vértice del cunjunto y recorrerá todos los vértices del grafo, si encuentra que uno de los nodos es adyacente con el nodo actual y el valor del vértice en la lista de valores más 1 es menor al valor actual del vértice de la iteración, actualizo el valor por el menor y repito. Una vez termine, armará el camino y lo retornará. **Complejidad**: $\mathcal{O}(N^2)$

V. Conclusiones

La realización de la estructura de datos `Graph` no fue en especial complicada, sin embargo, el desarrollo de los algoritmos fue una misión de destreza.

La mayoría de algoritmos que implementamos y analizamos resultan de carácter polinómico, sin embargo hay algunos algoritmos en la teoría de grafos que no pueden ser resueltos en tiempos polinómicos.

Además, experimentamos como la teoría de grafos es un amplio campo de estudio matemático en el cual las ciencias de la computación pueden explorar de una manera profunda.

Referencias

[1] West, D. B. (2005). Introduction to graph theory. New Delhi: Prentice-Hall of India Private Limited.

[2] A computer science portal for geeks. (n.d.). Retrieved from https://www.geeksforgeeks.org/