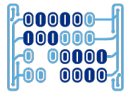

MACC
Matemáticas Aplicadas y
Ciencias de la Computación

Implementación de la clase `Graph` en C++

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

Víctor Samuel Pérez Díaz
Juan Esteban Murcia Nieto

21 Mayo 2019

Universidad del
Rosario

Esquema de presentación

Universidad del Rosario

Contextualización

- Nuestro **objetivo** es desarrollar una estructura de datos que nos permita implementar los algoritmos clásicos de la teoría de grafos no dirigidos.
- El lenguaje de programación empleado fue `C++` debido a su eficiencia y gran versatilidad.
- Nuestro código consta de tres partes:
  - **Interfaz** (`.hpp`)
  - **Implementación** (`.cpp`)
  - **Programa ejemplo** (`.cpp`)

Conceptos preliminares

La raiz de toda nuestra implementación es la clase `Graph`, la cual tiene como único parámetro la matriz de adyacencia (`matrix`) de un grafo G.
Esta está definida como:

Matriz de adyacencia

$$A(G) : a_{ij} = \begin{cases} 1 \text{ (Por arista)} & , v_i \leftrightarrow v_j, i \neq j \\ 0 & , v_i \not\leftrightarrow v_j, i \neq j \\ 2 \text{ (Por arista)} & , v_i \leftrightarrow v_j, i = j \end{cases}$$

Conceptos preliminares

Para lo anterior, se usó la clase `vector` de librería estándar de C++, la cual usamos para crear un vector de vectores, que nos serviría para representar la matriz.

Conceptos preliminares

Los **métodos** desarrollados fueron:

Métodos de la clase `Graph`

- `Graph` *constructor y sus derivados.*
- `uncolored_vertex`
- `colored_vertex`
- `colored_neighbor`
- `uv_path`
- `grade`
- `empty`
- `order`
- `size`
- `max_grade`
- `min_grade`

Conceptos preliminares

Los **métodos** desarrollados fueron:

Métodos de la clase `Graph`

- `remove_node`
- `remove_edge`
- `add_node`
- `add_edge`
- `clear`
- `print_matrix`
- `find_path`
- `color_graph`
- `breadth_search`
- `depth_search`
- `shortest_uv_path`

Funcionalidad

La clase `Graph` está basada en la matriz de adyacencia de un grafo no dirigido, esta siendo su único parámetro.
Un primer acercamiento fue creando dos estructuras `nodos (V)` y `aristas (E)`, pero resultaba un tanto conflictiva en la manipulación de la memoria.
La implementación representando el grafo mediante su matriz resultaba mucho más **práctica**.

Constructor `Graph`

Se desarrollaron varias versiones del constructor:

Constructores de la clase `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`.

Funciones Privadas

La clase cuenta con varias funciones las cuales no tienen un uso especial para el usuario, pero son cruciales para la buen funcionamiento de ciertos algoritmos, razón por la cual fueron definidas como privadas:

Funciones privadas

- *uncolored_vertex*: Dada una colaroción para el grafo, esta función determina si el nodo tiene o no color
- *colored_neighbor*: Dada una coloración para el grafo, esta función determina si un nodo tiiene algún vecino con un color específico
- *uv_path*: Función que determina si hay un uv-camino en el grafo

Funciones públicas

La clase le otorga al usuario diversas funciones con objetivos generales, cuyo objetivo es conocer y manipular el grafo que está trabajando.

Funciones Públicas

- `grade`: Retorna el grado de un nodo específico
- `empty`: Determina si el grafo está vacío
- `order`: Retorna la cantidad de nodos del grafo
- `size`: Retorna la cantidad de aristas del grafo
- `max_grade`: Retorna el grado máximo del grafo, $\Delta(G)$
- `min_grade`: Retorna el grado mínimo del grafo, $\delta(G)$
- `remove_node`: Elimina un nodo específico del grafo y todas sus aristas adyacentes
- `remove _edge`: Elimina una arista específica del grafo
- `add_node`: Añade un nodo al grafo, sin ninguna arista
- `add_edge`: Añade una arista entre 2 nodos específicos
- `clear`: Elimina todos los nodos y todas las aristas del grafo
- `print_matrix`: Imprime la matriz de adyacenciadel grafo

Algoritmos

find_path

Output: An Eulerian path
Initialization:
path,stk = empty
start, odd = 0
for *i in 1:n(G)* **do**
  if *grade(i) is odd* **then**
    odd += 1
    start = i
  end
end
if *odd > 2* **then**
  return empty
end
cur = start

Algorithm 1: Eulerian path

```
while stk not empty or garde(cur) > 0 do
    if grade(cur) == 0 then
        path.push(cur)
        cur = stk.top
        stk.pop
    end
    else
        for i=1:n(G) do
            if cur↔i then
                stk.push(cur)
                remove_edge(cur,i)
                cur = i
                break
            end
        end
    end
end
return path
```

color_graph

Output: A colored graph
out= (n,0)
clr = 1
while *Unolored_vertex* **do**
  for *i=1:n(G)* **do**
    if *i has no color and not colored_neighbor* **then**
      out[i] = clr
    end
  end
  clr += 1
end
return out

Algorithm 2: greedy coloring

shrtest_uv_path

Output: sortest uv path
values[i] = 0 if i=u, ∞ if not
Tm = nodes
path[i] = ∞
while *v in Tm* **do**
    vertex = i in Tm: values[i] is minimum
    erase i from Tm
    for *i in nodes* **do**
        if *vertex↔i and values[vertex]+1 < values[i]* **then**
            values[i] = values[vertex]+1
            path[i] = vertex
        end
    end
end

Algorithm 3: Dijkstras Algorithm

```
if path[v] = ∞ then
    return empty
end
push u to out
cur = v
while cur != u do
    insert cur in the 2^nd position of out
    cur = path[cur]
end
return out
```

También se implementaron algoritmos ya conocidos como:

- `breadth_search`
- `depth_search`

Ejemplo

Eejmplificaremos todos los algoritmos previamente explicados con el siguiente grafo:

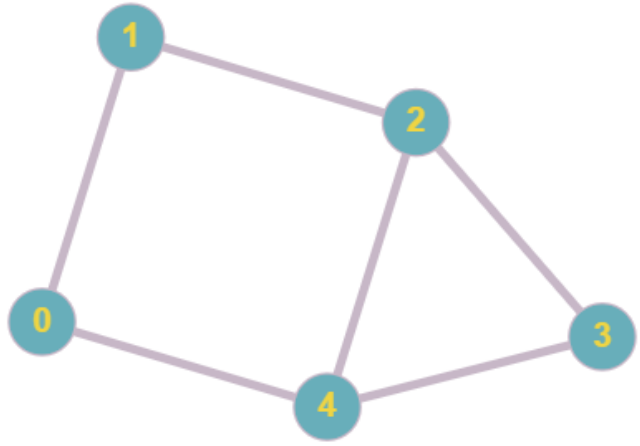

Figura: Grafo

 Douglas B. West
Introduction to graph theory.
New Delhi: Prentice- Hall of India Private Limited. (2005).

 Geeks for geeks
A computer science portal for geeks. (n.d.). Retrieved from https://www.geeksforgeeks.org/
Algorithms in graph theory.

¡Gracias!