Programa 5:
Búsqueda en Grafos mediante *Depth First Search (BFS)*

Estructura de Datos y Algoritmos II

Autor: José Mauricio Matamoros de Maria y Campos

Entrega: Lunes 30 de Marzo, 2020

1 Introducción

Objetivo: El estudiante conocerá e identificará las características necesarias para entender el algoritmo de búsqueda por profundidad o *Depth First Search (DFS)* en un grafo.

2 Antecedentes:

La búsqueda primero en profundidad o *Depth First Search (DFS)* es un algoritmo (o técnica) para recorrer grafos. Se utiliza para resolver problemas como los siguientes:

- En un grafo ponderado (con pesos), DFS produce el árbol de expansión mínimo y el árbol de camino más corto de todos los pares de vértices.
- Permite detectar ciclos en un grafo
- Permite realizar ordenaciones topológicas, operación que tiene aplicación en calendarización de tareas, evaluación de celdas en hojas de cálculo, síntesis lógica, etc.
- Resolver rompecabezas y laberintos de solución única.

Como su nombre lo indica, la estrategia del algoritmo de búsqueda en profundidad o *Depth First Search (DFS)* consiste en buscar "más profundo" dentro del grafo, siempre que sea posible. El algoritmo DFS explora las aristas no conocidas del último vértice descubierto. Una vez exploradas todas las aristas de v, el algoritmo "regresa" a explorar las aristas no exploradas del vértice precedente a v, es decir, el vértice a partir del cual se descubrió v. Este proceso continúa hasta que se hayan descubierto todos los vértices aaccesibles desde el vértice inicial. Si quedasen vértices no descubiertos, DFS seleccionará uno de estos como nuevo origen y repetirá la búsqueda desde allí, repitiendo el proceso hasta que se haya descubierto cada vértice[1]

Al igual que en BFS, cada vez que en DFS se descubre un vértice v al explorar la lista de adyacencia del vértice u previamente descubierto, la procedencia se almacena estableciendo la propiedad $v.\pi = u$. Sin embargo, a diferencia de BFS donde el subgrafo predecesor forma un árbol, el subgrafo predecesor producido en DFS puede estar compuesto por varios árboles, ya que la búsqueda puede repetirse desde múltiples fuentes. Por lo tanto, en DFS se define el subgrafo predecesor como:

$$G_\pi = (V, E_\pi)$$

dónde

$$E_\pi = \{(v, \pi, v) : v \in \text{V and } v_\pi \neq \text{NIL}\}$$

El subgrafo predecesor generado por DFS forma un **bosque de profundidad** formado por varios **árboles de profundidad**, donde las aristas en E_π son **aristas de árbol**.

[1]Puede parecer arbitrario que la búsqueda en amplitud o *BFS* se limite a un solo origen, mientras que la búsqueda de profundidad o *DFS* busca desde múltiples orígenes. Esto es debido a que BFS se utiliza por lo general para encontrar el camino más corto (y el subgrafo predecesor asociado) desde un punto dado, mientras que DFS es comúnmente una subrutina en otro algoritmo.

Al igual que en BFS, DFS "colorea" los vértices durante la exploración para indicar su estado. Cada vértice es inicialmente blanco, se colorea en gris al ser descubierto y se tiñe en negro cuando su lista de adyacencia ha sido completamente examinada. Así se garantiza que cada vértice forme parte de exactamente un árbol de profundidad, por lo que estos árboles son disjuntos.

Además de crear un bosque de profundidad, BFS añade marcas de tiempo o *timestamps* a cada vértice: $v.d$ cuando v es descubierto y coloreado en gris, y $v.f$ cuando v ha sido completamente explorado y coloreado en negro.

Estas marcas de tiempo proporcionan información importante sobre la estructura del grafo y, en general, son útiles para entender el comportamiento del algoritmo DFS.

El siguiente procedimiento DFS registra cuándo descubre el vértice u en el atributo u: d y cuando termina el vértice u en el atributo u.f. Dichas marcas son enteros entre 1 y $2\,|V|$, debido a que sólo se registra un evento de descubrimiento y un evento de finalización para cada uno de los vértices $|V|$. Así, para cada vértice u,

$$u.d < u.f$$

El vértice u es WHITE (blanco) antes del tiempo $u.d$, GRAY (gris) entre $u.d$ y $u.f$, y posteriormente BLACK (negro).

El siguiente pseudocódigo corresponde al algoritmo básico DFS, donde el grafo G puede ser dirigido o no y la variable de tiempo es de ámbito global.

```
procedure DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    time = 0
    for each vertex u ∈ G.V do
        if u.color == WHITE then
            DFS-VISIT(G, u)
        end if
    end for
end procedure
```

```
procedure DFS-VISIT(G, u)
    time = time + 1                          ▷ Descubre el vércice blanco u
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj [u] do                ▷ Explora la arista (u, v)
        if v.color == WHITE then
            v.π = u
            DFS-VISIT(G, v)
        end if
    end for                                  ▷ Colorea u en negro al terminar
    u.color = BLACK
    time = time + 1
    u.f = time
end procedure
```

3 Desarrollo:

Con base en el código del Apéndice A, realice los apartados que se muestran en este documento.

- [1 punto] Ejecute el programa `graph.py` con el archivo de ejemplo `example.graph` y anote la salida del mismo. Dibuje el grafo obtenido.

- [1 punto] ¿En qué difiere la ejecución de la función DFS de la función BFS realizada en el programa anterior?
- [4 puntos] Modifique la función `dfs` del programa `graph.py` para que calcule el número de islas de un grafo y pruébelo con el grafo almacenado en `example.graph`.
  Nota: Se dice que un grafo tiene islas cuando existe al menos un vértice v para los cuales no existe una ruta desde un vértice u. Así, todos los vértices accesibles desde v forman una isla y todos los vértices accesibles desde u forman otra isla (véase Figura 1).

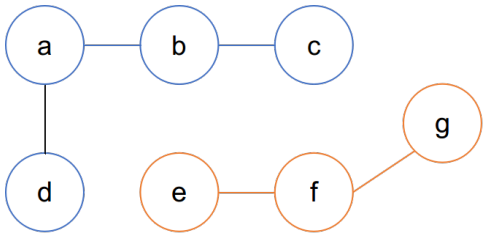

Figure 1: Grafo con dos islas

- [4 puntos] Modifique la función `dfs` del programa `graph.py` para que calcule el número de ciclos de un grafo y pruébelo con el grafo almacenado en `example.graph`.
  Nota: Se dice que un grafo tiene ciclos cuando existe más de una ruta de un vértice u a un vértice v (véase Figura 2).

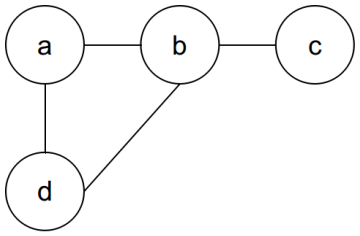

Figure 2: Grafo con un ciclo

A Archivo `graph.py`

graph.py

```
class Vertex:
  # Constructor
  def __init__(self, name, neighbors = []):
    self._name = name
    # list of neighbors of this vertex (aka edges)
    self._neighbors = list(neighbors)
    # Misc info used only by the bfs algorithm
    # self.color = 'WHITE'
    # self.d = -1
    # self.p = None
  #end def

  @property
  def name(self):
    return self._name
  #end def

  @property
  def neighbors(self):
    return self._neighbors
  #end def

  def __str__(self):
    return '{} = {}'.format(self._name, ', '.join([n.name for n in self._neighbors]))
  #end def

  def __repr__(self):
    info = ''
    if 'color' in self.__dict__:
      info = ' ({}, d={}, p={})'.format(self.color, self.d, self.p.name if self.p != None
    else '')
    return '{}{} = {}'.format(self._name, info, ', '.join([n.name for n in self._neighbors])
    )
  #end def
#end class


class Graph:
  # Constructor
  def __init__(self, vertexes = [] ):
    self._vertexes = list(vertexes)
  #end def

  @property
  def vertexes(self):
    return self._vertexes
  #end def

  def __getitem__(self, key):
    matches = [ vertex for vertex in self.vertexes if vertex.name == key ]
    if len(matches) == 1:
      return matches[0]
    elif len(matches) > 1:
      return matches
    return None

  #end def

  def __str__(self):
    return '\n'.join([ v.__str__() for v in self._vertexes ])
  #end def

  def __repr__(self):
    return '\n'.join([ v.__repr__() for v in self._vertexes ])
```

```
63    #end def
64
65    # Loads a graph from an adjacency-list file
66    @staticmethod
67    def from_file( file_path ):
68      graph = Graph()
69      vertexes = {}
70      line_num = 0
71      with open(file_path, 'r') as fp:
72        line = fp.readline()
73        while line:
74          # Each line contains a comma-separated list of neighbors
75          # e.g. a =  b, c, d
76          parts = re.split(r'\s*[\:\=]\s*', line.strip())
77          vertex_name = parts[0]
78          vertex_nix = re.split(r'\s*,\s*', parts[1])
79          if vertex_name in vertexes:
80            raise Exception('Duplicated vertex {} declared in line {}'.format(vertex_name,
      line_num))
81            vertexes[vertex_name] = (Vertex(vertex_name), vertex_nix)
82            line_num += 1
83            line = fp.readline()
84      # Perform a concistency check after loading the adjacency list
85      # While doing it, the object is structured
86      for pair in vertexes.values():
87        vobj, vnix = pair
88        for n in vnix:
89          if not n in vertexes:
90            raise Exception('Vertex {} is connected to {} but was not declared (incomplete
      graph)'.format(n, v))
91            vobj.neighbors.append(vertexes[n][0])
92        graph.vertexes.append(vobj)
93      # Finally we return the graph object
94      return graph
95    #end def
96  #end class
97
98  time = 0
99  def dfs(graph):
100   global time
101   time = 0
102
103   def dfs_visit(graph, u):
104     global time
105     time = time + 1
106     u.d = time
107     u.color = 'GRAY'
108     for v in u.neighbors:
109       if v.color == 'WHITE':
110         v.p = u
111         dfs_visit(graph, v)
112     u.color = 'BLACK'
113     time = time + 1
114     u.f = time
115   #end dfs-visit
116
117   for u in graph.vertexes:
118     u.color = 'WHITE'
119     u.p = None
120
121   for u in graph.vertexes:
122     if u.color == 'WHITE':
123       dfs_visit(graph, u)
124
125 #end def
126
127 def main():
128   graph = Graph.from_file('example.graph')
```

```
129 print(graph)
130 dfs(graph)
131 print(repr(graph))
```