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

Estructura de Datos y Algoritmos II

Autor: José Mauricio Matamoros de Maria y Campos

Entrega: Lunes 23 de Marzo, 2020

## 1 Introducción

**Objetivo:** El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para entender el algoritmo *Breadth First Search (BFS)* o búsqueda por expansión.

## 2 Desarrollo:

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

- [1 punto] Modifique el programa `graph.py` para que cargue el archivo `example.graph` e imprima dicho grafo. Dibuje el grafo obtenido.
- [1 punto] Modifique el programa `graph.py` para que cargue el archivo `metro.graph` y calcule la ruta entre las estaciones *Zapata* y *Portales*. Anote la ruta obtenida.
- [8 puntos] Modifique la función `bfs` del programa `graph.py` para que imprima el número de operaciones requeridas para generar el grafo de ruta óptima y con los resultados obtenidos complete la Cuadro 1:

Table 1: Tiempo de búsqueda con diferentes tamaños de tabla hash

| Origen | Destino | $\Theta(E, V)$ | Ruta |
|---|---|---|---|
| Zapata | Portales | | |
| Guerrero | Chabacano | | |
| Obrera | Tlatelolco | | |
| Hangares | Observatorio | | |
| Universidad | Ciudad Azteca | | |

# A Archivo **graph.py**

graph.py

```python
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 ])
```

```python
  #end def

  # Loads a graph from an adjacency-list file
  @staticmethod
  def from_file( file_path ):
    graph = Graph()
    vertexes = {}
    line_num = 0
    with open(file_path, 'r') as fp:
      line = fp.readline()
      while line:
        # Each line contains a comma-separated list of neighbors
        # e.g. a =  b, c, d
        parts = re.split(r'\s*[\:\=]\s*', line.strip())
        vertex_name = parts[0]
        vertex_nix = re.split(r'\s*,\s*', parts[1])
        if vertex_name in vertexes:
          raise Exception('Duplicated vertex {} declared in line {}'.format(vertex_name, line_num))
        vertexes[vertex_name] = (Vertex(vertex_name), vertex_nix)
        line_num += 1
        line = fp.readline()
    # Perform a concistency check after loading the adjacency list
    # While doing it, the object is structured
    for pair in vertexes.values():
      vobj, vnix = pair
      for n in vnix:
        if not n in vertexes:
          raise Exception('Vertex {} is connected to {} but was not declared (incomplete graph)'.format(n, v))
        vobj.neighbors.append(vertexes[n][0])
      graph.vertexes.append(vobj)
    # Finally we return the graph object
    return graph
  #end def
#end class

def bfs(graph, start):
  for u in graph.vertexes:
    if u == start:
      continue
    u.color = 'WHITE'
    u.d = -1
    u.p = None
  start.color = 'GRAY'
  start.d = 0
  start.p = None
  queue = Queue()
  queue.enqueue(start)
  while len(queue) > 0:
    u = queue.dequeue()
    for v in u.neighbors:
      if v.color != 'WHITE':
        continue
      v.color = 'GRAY'
      v.d = u.d + 1
      v.p = u
      queue.enqueue(v)
    # end for
    u.color = 'BLACK'
  #end while
# end def

def best_route(graph, start, end):
  bfs(graph, start)
  return get_path(graph, start, end)

def get_path(graph, start, end):
```

```python
  if start == end:
    return start.name
  elif end.p == None:
    return "There's no path from {} to {}".format(start, end)
  else:
    return '{} -> {}'.format(get_path(graph, start, end.p), end.name)


def main():
  graph = Graph.from_file('metro.graph')
  print(best_route(graph, graph['Copilco'], graph['Portales']))
```