Árbol de binario y algoritmo de búsqueda en anchura en Python

José Alberto Benavides Vázquez

15 de abril de 2018

En este reporte se pretende demostrar el funcionamiento de un algoritmo de búsqueda en anchura a partir de un árbol binario desarrollado mediante un paquete para construir grafos alojado en `https://github.com/jbenavidesv87/FlujoRedes`.

Un **grafo** es una representación mediante círculos u otras figuras, llamados **nodos**, de sistemas que cuentan con elementos o grupos de elementos que están relacionados entre sí. Para denotar estas relaciones entre los elementos, se pueden usar colores, posiciones y líneas que van de un nodo a otro, conocidas como **arcos**, mismas que pueden tener asignados valores numéricos, denominados **pesos**. Además, los nodos de los grafos pueden estar dispuestos de maneras ordenadas, como formando una circunferencia. Un ejemplo de esto puede verse en la figura 1 (p. 2).

Un grafo en el que cada nodo puede estar solamente conectado a otros dos, de manera secuencial, es denominado árbol binario. A cada par de nodos conectados otro, se les denomina **hijos**, mientras que al nodo que se conectan se le llama **padre**. En ciertas aplicaciones computacionales, los árboles binarios son usados frecuentemente porque constituyen una manera eficiente de estructurar datos[1]. Mediante el paquete de desarrollo de grafos antes mencionado, se desarrolló un programa que genera árboles binarios dada una cantidad de ramas o capas. El código usado es el siguiente:

```
1   from math import floor
2
3   from Grafo import Grafo
4   from Nodo import Nodo
5
6   G = Grafo()
7
8   niveles = 5
9   hijos = 2
10  maxNodosX = hijos ** niveles
11  radioNodo = 1 / maxNodosX / 3
12  yDesp = 1 / niveles
13
14  nodo = Nodo()
15  nodo.id = ""
16  nodo.posicion = (0.5, 1)
```

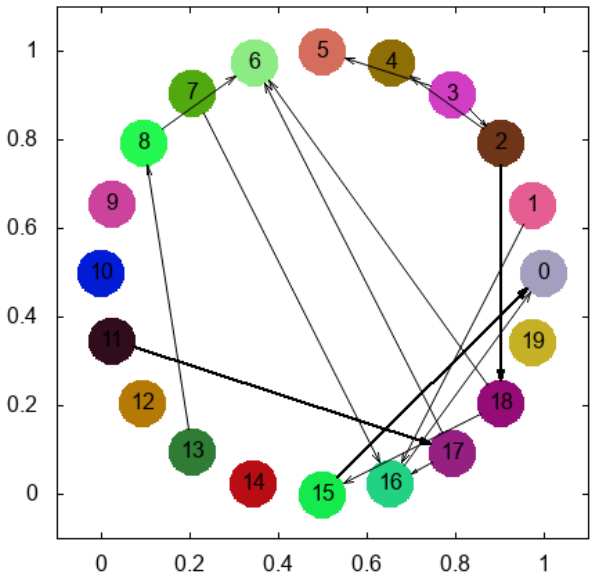

Figura 1: Grafo con nodos dispuestos de manera circular, con algunos nodos unidos entre sí con arcos en color negro.

```
17      nodo.radio = radioNodo
18      G.AgregarNodo(nodo)
19
20      for nivel in range(1, niveles + 1):
21          hijosNivel = hijos ** nivel
22          xInicial = 1 / (hijosNivel * 2)
23          for hijo in range(0, hijosNivel):
24              n = Nodo()
25              n.id = ""
26              n.posicion = (xInicial + 1 / (hijosNivel) * hijo, 1 -
    yDesp * nivel)
27              n.radio = radioNodo
28              G.ConectarNodos(G.nodos[hijos ** (nivel - 1) - 1 +
    floor(hijo / hijos)], n)
29
30      G.BreadthFirst(nodo, 0, True)
31
```

En este programa, se crean 2^n nodos por cada nivel del árbol, de un total de niveles definido en la línea 8. En la línea 28 se encuentra la función que conecta cada nodo con su padre correspondiente. El grafo resultante para 5 niveles se muestra en la figura 3 (p. 4).

El algoritmo de búsqueda en anchura consiste en recorrer, a partir de un nodo inicial denominado s, todos sus vecinos y luego, de cada vecino, recorrer sus vecinos y así recursivamente. Se llama **visitar** al hecho de que el algoritmo recorra determinado nodo. Una peculiaridad de este algoritmo, es que un nodo

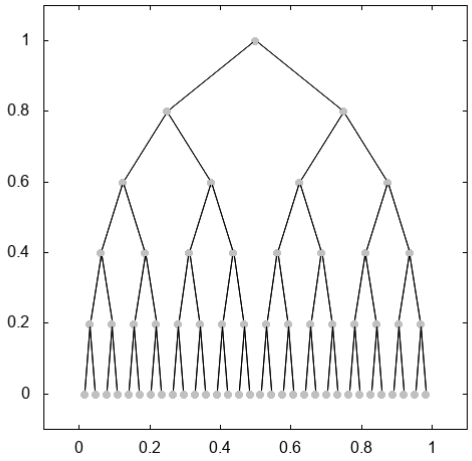

Figura 2: Grafo del árbol binario de 5 niveles con que se probará el algoritmo de búsqueda en anchura.

sólo puede ser visitado una sola vez. En este caso, se eligió como parámetro para marcar a un nodo visitado el uso de color rojo. Así, el algoritmo sólo visita nodos de un color distinto al rojo, por lo que inicialmente se colorearon de gris los nodos de este ejemplo. El algoritmo desarrollado genera una imagen `png` cada vez que el algoritmo visita a un nodo y, finalmente, las imágenes resultantes de cada visita fueron agrupadas en una animación `gif`. La animación puede verse en `FALTA` y una secuencia de 4 imágenes del algoritmo en la figura **??** (p. **??**).

El código del algoritmo, implementado en el paquete mencionado es el siguiente:

```
    def BreadthFirst(self, s, i, debug = False):
      s.Color(255, 0, 0);
      if debug:
          self.nombre = "{:06d}".format(i)
          self.DibujarGrafo(titulo = str(i))
          remove('{:06d}.gnu'.format(i)) # https://pyformat.info/
        i = i + 1
      for v in self.vecinos[s]:
          if v.color != "#00ff0000":
              if debug:
                  i = self.BreadthFirst(v, i, True)
              else:
                  self.BreadthFirst(v, i)
      return i

```

Este método, llamado `BreadthFirst`, se incorporó dentro de la clase `Grafo` del paquete y recibe los siguientes parámetros:

1. `self`: El grafo sobre el que se ejecuta el método.

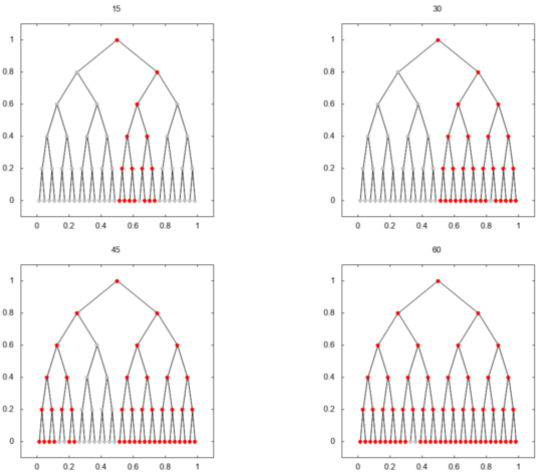

Figura 3: Imágenes de las visitas 15, 30, 45 y 60 del grafo de ejemplo para este reporte.

2. `s`: Nodo a partir del cual se comienza la búsqueda.
3. `i`: Una variable entera utilizada para contar el número de veces que se visita un nodo por primera vez.
4. `debug`: Una variable booleana.

En su primera iteración, se colorea el nodo `s` del que se parte (línea 2); si la variable `debug` es verdadera, se genera una imagen `png` correspondiente al grafo resultante de esta iteración (líneas 3 a 6); se suma 1 al iterador `i` (línea 7); luego, por cada vecino del nodo `s`, si su color es distinto a rojo (línea 9), se vuelve a llamar este método pasando como parámetro `s` al vecino en turno (líneas 10 a la 13). Finalmente, el algoritmo regresa `i`, para acumular las iteraciones recursivas que se hayan contado en caso de haber sido necesarias.

Referencias

[1] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1969.