Universidad de San Carlos de Guatemala
Facultad de ingeniería
Escuela de ciencias
Departamento de matemática
Matemática para Computación 2, sección A
Segundo semestre 2023
Ing. José Alfredo González Díaz
Aux. Edgar Daniel Cil Peñate

Proyecto de Investigación:

Algoritmos de Búsqueda en Árboles por Ancho y por Profundidad.

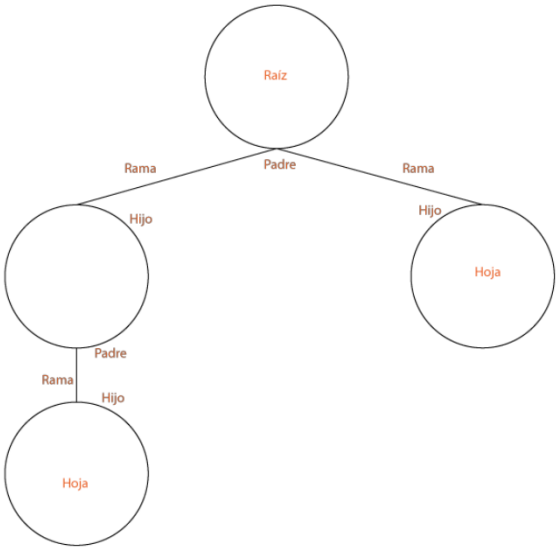
Juan José Roberto Carranza López

Carné 200819087

Introducción

En programación es muy común usar Estructuras de Datos para poder ordenar y almacenar información. Ésta puede ser desde primitivas como números enteros hasta cadenas de texto hasta tipos definidos por el programador como productos, personas o estados de un juego. Existen diversas estructuras de datos entre las cuales están los arrays, las listas, las colas, las pilas, los conjuntos, los árboles, los grafos, etc. En este proyecto vamos a centrarnos en los árboles y en los dos algoritmos de búsqueda más sencillos que podemos usar en ellos: la búsqueda en anchura y la búsqueda en profundidad.

Sabemos que un árbol es una estructura de datos que consta de nodos y ramas. Cada nodo representa un elemento que deseamos organizar, y las ramas son conexiones dirigidas entre pares de nodos. Un nodo se considera padre si está conectado por una rama a otro nodo, y se llama hijo si existe una rama que lo conecta con otro nodo. Cuando un nodo no tiene hijos, se denomina hoja. Un grafo es una extensión o generalización de un árbol.



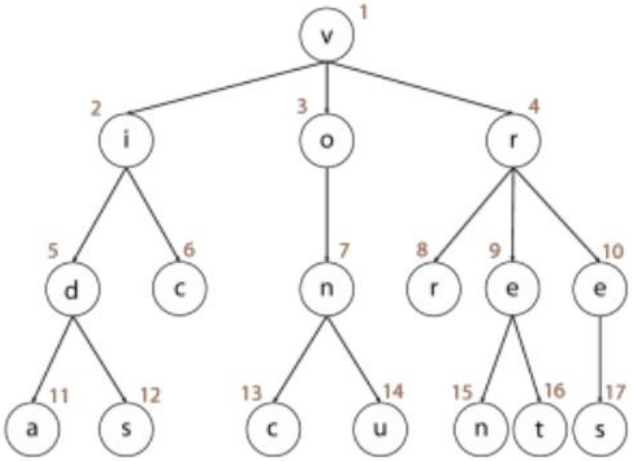
"Como cualquier estructura de datos, los árboles presentan tres operaciones fundamentales: la inserción de nuevos elementos, la eliminación de elementos existentes y la búsqueda de un elemento en la estructura. Existen otras operaciones, como la ordenación y la mezcla de dos estructuras de datos del mismo tipo, pero estas no serán abordadas en este proyecto.

La próxima etapa implicará la creación de las búsquedas en anchura y profundidad. Sin embargo, antes de proceder con su implementación, es necesario comprender el propósito de cada una de ellas.

Búsqueda en Árboles por Ancho

La búsqueda en anchura, conocida en inglés como Breadth First Search, es un algoritmo empleado para explorar o buscar elementos en estructuras de datos como árboles y grafos (aunque en este proyecto nos enfocaremos en árboles). Pertenece al grupo de las búsquedas no informadas (sin heurísticas).

Su procedimiento consiste en ir visitando todos los nodos de un nivel antes de proceder con el siguiente nivel tal y como mostramos en la siguiente figura (los números indican el orden de exploración de los nodos):



Entonces, ¿cómo logramos que la búsqueda en anchura funcione? La respuesta es bastante simple: Utilizamos una cola como una estructura de datos auxiliar. Una cola es una estructura FIFO (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. En este contexto, comenzamos visitando la raíz, luego los hijos de la raíz, seguidos de los hijos de cada uno de estos hijos, y así sucesivamente.

Dado que hemos seleccionado Java como lenguaje de programación para este proyecto, podemos aprovechar la interfaz Queue y la clase LinkedList para aprovechar la funcionalidad de las colas sin necesidad de escribir código adicional.

Nuestro procedimiento va a ser el siguiente:

- Insertamos la raíz en la cola, como preparación.
- Si la cola no está vacía, sacamos el primer elemento (el primer nodo que haya en la cola) y comprobamos si es el elemento que estamos buscando. Si es igual entonces acabamos, si no es igual obtenemos todos los hijos de dicho nodo y los insertamos en la cola (recordamos que las inserciones son por el final).
- Repetimos hasta que hayamos encontrado el elemento o la cola sea vacía.
- Finalizamos.

Como la búsqueda se realiza desde un NodoArbol (la raíz), cabe pensar que la función de búsqueda debe ser parte de esta clase. Por tanto, agregamos la siguiente función a la clase NodoArbol:

```
1  public boolean busquedaAnchura(char c) {
2     Queue<NodoArbol> colaAuxiliar = new LinkedList<NodoArbol>();
3     colaAuxiliar.add(this);
4
5     while(colaAuxiliar.size() != 0) {
6        NodoArbol cabeza = colaAuxiliar.poll();
7        System.out.println(cabeza.getElemento());  // solo añadido como
  informacion para nosotros
8        if(cabeza.getElemento() == c)
9           return true;
10       else
11          for(NodoArbol hijo : cabeza.getHijos())
12             colaAuxiliar.add(hijo);
13    }
14    return false;
15 }
```

Como hemos mencionado previamente, lo primero es insertar la raíz en la cola para que tengamos algo por dónde empezar. A partir de entonces, si existe algo en la cola tomamos la cabecera (la función poll() devuelve la cabeza y la elimina de la cola). Ahora comprobamos si el elemento interno de este nodo es el que estamos buscando (recordemos que se debe usar la función .equals() para comparar objetos no primitivos). Si el elemento es el que buscamos entonces retornamos de la función positivamente, si no lo es entonces cogemos todos los hijos de dicho nodo y los insertamos en la cola para hacer una iteración más.

Llega el momento de probar que lo que hemos hecho funciona como esperamos. Los siguientes códigos se deben insertar al final del main de la clase ArbolesBusquedas y lo que obtendría

```
1  System.out.println(raiz.busquedaAnchura('v'));
2
3  /* output
4  v
5  true
6  */
```

Comprobamos que encontramos la letra ‘u’:

```
System.out.println(raiz.busquedaAnchura('u'));

/* output
v
i
o
r
d
c
n
r
e
e
a
s
c
u
true
*/
```

Comprobamos que no encontramos la letra ‘z’:

```
System.out.println(raiz.busquedaAnchura('z'));

/* output
v
i
o
r
d
c
n
r
e
e
a
s
c
u
n
t
s
false
*/
```

¿Qué pasa si buscamos la letra ‘c’? Existen varias letras ‘c’, ¿cuál encontrará? Probemos:

```
System.out.println(raiz.busquedaAnchura('c'));


/* output
v
i
o
r
d
c
true
*/
```

Entonces, la búsqueda en anchura va a encontrar el elemento de menor profundidad del árbol, si es que existe.

Ventajas de la búsqueda en anchura:

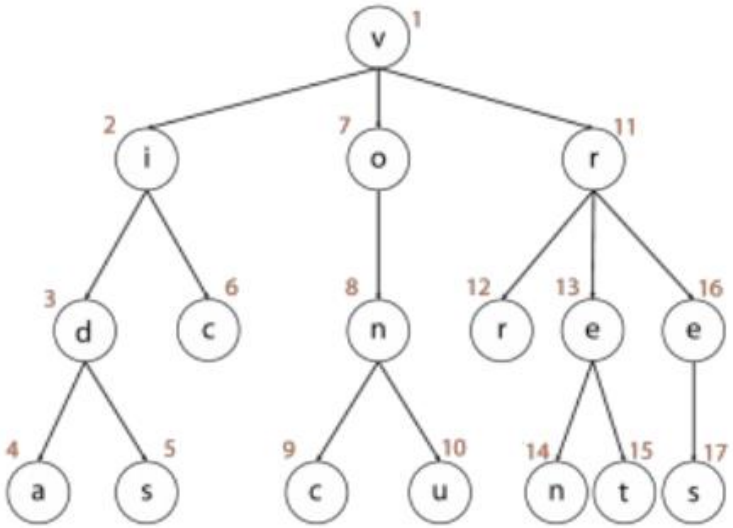
- Es completo: siempre encuentra la solución si existe.
- Es óptimo si el coste de cada rama es constante: en Inteligencia Artificial puede que cada nodo sea un estado de un problema, y que unas ramas tengan un coste diferente a las demás.

Inconvenientes de la búsqueda en anchura:

- Complejidad exponencial en espacio y tiempo (incluso peor la del espacio que la del tiempo).

Búsqueda en Árboles por Profundidad

La búsqueda en profundidad, llamada Depth First Search en inglés, es un algoritmo usado para recorrer o buscar elementos en un árbol o un grafo y pertenece al grupo de las búsquedas no informadas (sin heurísticas). Su procedimiento consiste en visitar todos los nodos de forma ordenada pero no uniforme en un camino concreto, dejando caminos sin visitar en su proceso. Una vez llega al final del camino vuelve atrás hasta que encuentra una bifurcación que no ha explorado, y repite el proceso hasta acabar el árbol (esto se conoce como backtracking). En la siguiente figura mostramos el orden de visita, siendo los números en naranja dicho orden:



Para poder programar dicho comportamiento vamos a apoyarnos en una estructura de datos llamada pila. Una pila es una estructura de datos LIFO (Last In, First Out) en la que sólo disponemos de dos operaciones: apilar y desapilar. Apilar añade el elemento en la cabeza y desapilar lo extrae y elimina. Por tanto, el primer elemento que entre se quedará en el fondo y será el último en visitarse. Igual que con las colas, Java nos brinda la clase Stack que nos va a servir el comportamiento deseado.

Nuestro procedimiento, por tanto, será:

- Insertamos la raíz en la pila, para preparación.
- Mientras haya algo en la pila, desapilamos la cabeza. Comprobamos si es el elemento que buscamos y si lo es acabamos. Si no lo es, cogemos todos los hijos de dicho nodo y los apilamos todos.
- Repetimos hasta que encontremos el elemento buscado o la pila esté vacía.
- Finalizamos.

Por tanto, añadimos la siguiente función a la clase NodoArbol:

```
public boolean busquedaProfundidad(char c) {
   Stack<NodoArbol> pilaAuxiliar = new Stack<NodoArbol>();
   pilaAuxiliar.push(this);

   while(pilaAuxiliar.size() != 0) {
      NodoArbol cabeza = pilaAuxiliar.pop();
      System.out.println(cabeza.getElemento());  // solo añadido como
informacion para nosotros
      if(cabeza.getElemento() == c)
         return true;
      else
         for(int i = cabeza.getHijos().size() - 1; i >= 0; i--)
            pilaAuxiliar.push(cabeza.getHijos().get(i));
   }
   return false;
}
```

La razón por la que no hemos utilizado un bucle for similar al de la búsqueda en anchura es que deseamos explorar el árbol en un orden específico, es decir, de izquierda a derecha.

En una pila, la estructura es LIFO (Last In, First Out), lo que significa que el último elemento en entrar es el primero en salir. Dado que queremos recorrer el árbol por el camino de más a la izquierda, necesitamos que el hijo de más a la izquierda del nodo actual sea el siguiente en ser consultado. Si utilizáramos el mismo bucle for que en la búsqueda en anchura, siempre estaríamos consultando al hijo de más a la derecha del nodo actual, lo que no estaría en línea con nuestro objetivo de recorrer el árbol de izquierda a derecha. Por lo tanto, adaptamos el enfoque utilizando una pila para lograr el recorrido deseado.

Ahora procedamos a verificar su funcionamiento. Como en los pasos anteriores, agregaremos los siguientes códigos al final de la creación del árbol en la función principal. Al ejecutarlos, podremos observar los resultados que están marcados como output:

Comprobamos que encuentra la raíz:

```
System.out.println(raiz.busquedaProfundidad('v'));


/* output
v
true
*/
```

Comprobamos que encuentra la letra ‘u’:

```
System.out.println(raiz.busquedaProfundidad('u'));

/* output
v
i
d
a
s
c
o
n
c
u
true
*/
```

Comprobamos que no encuentra la letra ‘z’:

```
System.out.println(raiz.busquedaProfundidad('z'));

/* output
v
i
d
a
s
c
o
n
c
u
r
r
e
n
t
e
s
false
*/
```

¿Qué pasa si buscamos la letra ‘c’?

```
System.out.println(raiz.busquedaProfundidad('c'));

/* output
v
i
d

a
s
c
true
*/
```

Como vemos, la búsqueda en profundidad busca el elemento por el camino de máxima profundidad y cuando éste se acaba, vuelve al último nodo que había visitado con caminos posibles (caminos abiertos).

Ventajas de la búsqueda en profundidad:

- Es completa si no existen ciclos repetidos.
- Tiene menor complejidad en espacio que la búsqueda en anchura, porque solo mantenemos en memoria un camino simultáneamente.

Inconvenientes de la búsqueda en profundidad:

- No es óptima.
- Puede no encontrar la solución, aunque exista si hay caminos infinitos. Luego no es completa.

Concluiremos con que en realidad no hay una búsqueda mejor o peor, porque dependen mucho de la posición de nuestra solución dentro del árbol (entendemos solución por lo que estamos buscando).

Algunos usos de las búsquedas

Los algoritmos de búsqueda en árboles por ancho y por profundidad tienen una gran variedad de aplicaciones en la ciencia. Algunos ejemplos incluyen:

- Búsqueda de información: Estos algoritmos se pueden utilizar para buscar información en árboles de datos, como árboles binarios de búsqueda o árboles de decisión. Por ejemplo, un algoritmo de búsqueda por ancho se puede utilizar para buscar un producto en un catálogo de productos, mientras que un algoritmo de búsqueda por profundidad se puede utilizar para diagnosticar un problema en un sistema de software.

- Cálculo de rutas: Estos algoritmos se pueden utilizar para calcular rutas en árboles de datos, como árboles de grafos o árboles de rutas. Por ejemplo, un algoritmo de búsqueda por ancho se puede utilizar para encontrar la ruta más corta entre dos ciudades, mientras que un algoritmo de búsqueda por profundidad se puede utilizar para encontrar todos los caminos posibles entre dos nodos en un árbol.

- Juegos: Estos algoritmos se pueden utilizar para jugar juegos de árboles, como el ajedrez o el Go (Juego de mesa, de origen chino, de estrategia abstracta para dos jugadores en el que el objetivo es rodear más territorio que el oponente). Por ejemplo, un algoritmo de búsqueda por ancho se puede utilizar para explorar todas las posibilidades en un juego de ajedrez, mientras que un algoritmo de búsqueda por profundidad se puede utilizar para encontrar el mejor movimiento en un juego de Go.

Aquí hay algunos ejemplos específicos de cómo se utilizan los algoritmos de búsqueda en árboles por ancho y por profundidad en la ciencia:

- En el ámbito de la informática, los algoritmos de búsqueda en árboles se utilizan para implementar una gran variedad de estructuras de datos y algoritmos, como las listas ligadas, las pilas y las colas.

- En el ámbito de la biología, los algoritmos de búsqueda en árboles se utilizan para analizar árboles filogenéticos, que representan las relaciones evolutivas entre las especies.

- En el ámbito de la medicina, los algoritmos de búsqueda en árboles se utilizan para diagnosticar enfermedades.

- En el ámbito de la ingeniería, los algoritmos de búsqueda en árboles se utilizan para diseñar circuitos electrónicos.

- En general, los algoritmos de búsqueda en árboles son herramientas poderosas que se pueden utilizar para resolver una amplia gama de problemas en la ciencia.

Ejemplo de Aplicación de Búsqueda en Árboles por Ancho.

Distancia y caminos cortos

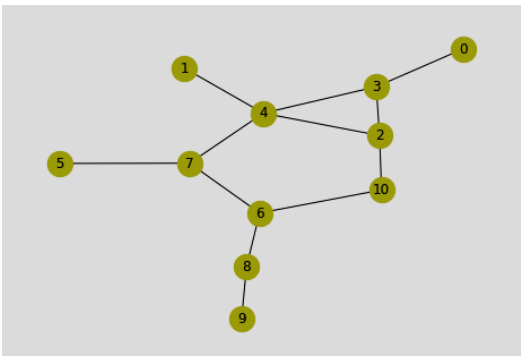
Tomemos una gráfica conexa ***G*** y dos vértices ***u*** y ***v***, supongamos que queremos encontrar la distancia de ***v*** a ***u*** de manera algorítmica. Una forma de resolver esto es hacer una búsqueda por anchura que comience en ***v***. Es sencillo ver que, si ***T*** es el árbol por anchura asociado, entonces la distancia buscada es la cantidad de veces que debemos aplicar la función padre a ***u*** para llegar a ***v***.

Esta idea nos lleva a la siguiente implementación.

ejemplo_1_busqueda_ancho.py > distancia

```python
# Aplicación de búsqueda en Árboles por Ancho
# Distancia y caminos cortos

import networkx as nx

def distancia(G,v,u):
    if v==u:
        return 0
    encontrados={v}
    distancias={v:0}
    procesados=set()
    en_proceso=[v]
    while en_proceso:
        w=en_proceso[0]
        for z in G.neighbors(w):
            if (z not in procesados) and (z not in encontrados):
                encontrados.add(z)
                en_proceso.append(z)
                distancias[z]=distancias[w]+1
                if z==u:
                    return(distancias[z])
        en_proceso.remove(w)
        procesados.add(w)
    return "No están conectados, la gráfica no es conexa."

G = nx.Graph()
G.add_edges_from([(0,3), (1,4), (7,4), (10,2), (2,3), (2,4), (3,4), (6,7), (5,7), (6,8), (8,9), (6,10)])
nx.draw_kamada_kawai(G,with_labels=True, node_color='#bbbb22',node_size=500)

print(distancia(G,1,10))
print(distancia(G,5,0))
print(distancia(G,0,4))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\WINDOWS\System32\WindowsPowerShell\v1.0> & C:/Users/el/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/el/Docum
uto 2/Proyecto/[MC2]Proyecto_200819087/ejemplo_1_busqueda_ancho.py"
3
4
2
PS C:\WINDOWS\System32\WindowsPowerShell\v1.0>
```



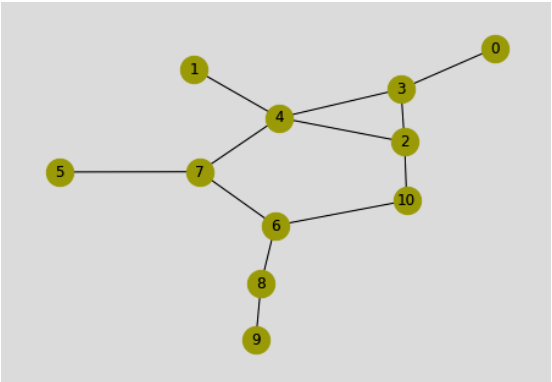
Si en el algoritmo anterior llevamos registro de los padres de cada vértice, entonces podemos no sólo determinar la distancia de ***v*** a ***u***, sino además encontrar un camino más corto entre ellos.

ejemplo_2_busqueda_ancho.py > ...

```python
# Este calcula explícitamente uno de los caminos más cortos de v a u

import networkx as nx

G = nx.Graph()
G.add_edges_from([(0,3), (1,4), (7,4), (10,2), (2,3), (2,4), (3,4), (6,7), (5,7), (6,8), (8,9), (6,10)])

def camino_corto(G,v,u):
    if v==u:
        return [u]
    encontrados={v}
    procesados=set()
    padres={v:None}
    en_proceso=[v]
    while en_proceso:
        w=en_proceso[0]
        for z in G.neighbors(w):
            if (z not in procesados) and (z not in encontrados):
                encontrados.add(z)
                en_proceso.append(z)
                padres[z]=w
                if z==u:
                    camino=[z]
                    padre=z
                    while padre!=v:
                        padre=padres[padre]
                        camino=[padre]+camino
                    return camino
        en_proceso.remove(w)
        procesados.add(w)
    return "No están conectados"

nx.draw_kamada_kawai(G,with_labels=True, node_color='#bbbb22',node_size=500)

print(camino_corto(G,1,10))
print(camino_corto(G,5,0))
print(camino_corto(G,0,4))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS C:\WINDOWS\System32\WindowsPowerShell\v1.0> & C:/Users/el/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/el/Docume
uto 2/Proyecto/[MC2]Proyecto_200819087/ejemplo_2_busqueda_ancho.py"
[1, 4, 2, 10]
[5, 7, 4, 3, 0]
[0, 3, 4]
PS C:\WINDOWS\System32\WindowsPowerShell\v1.0>
```



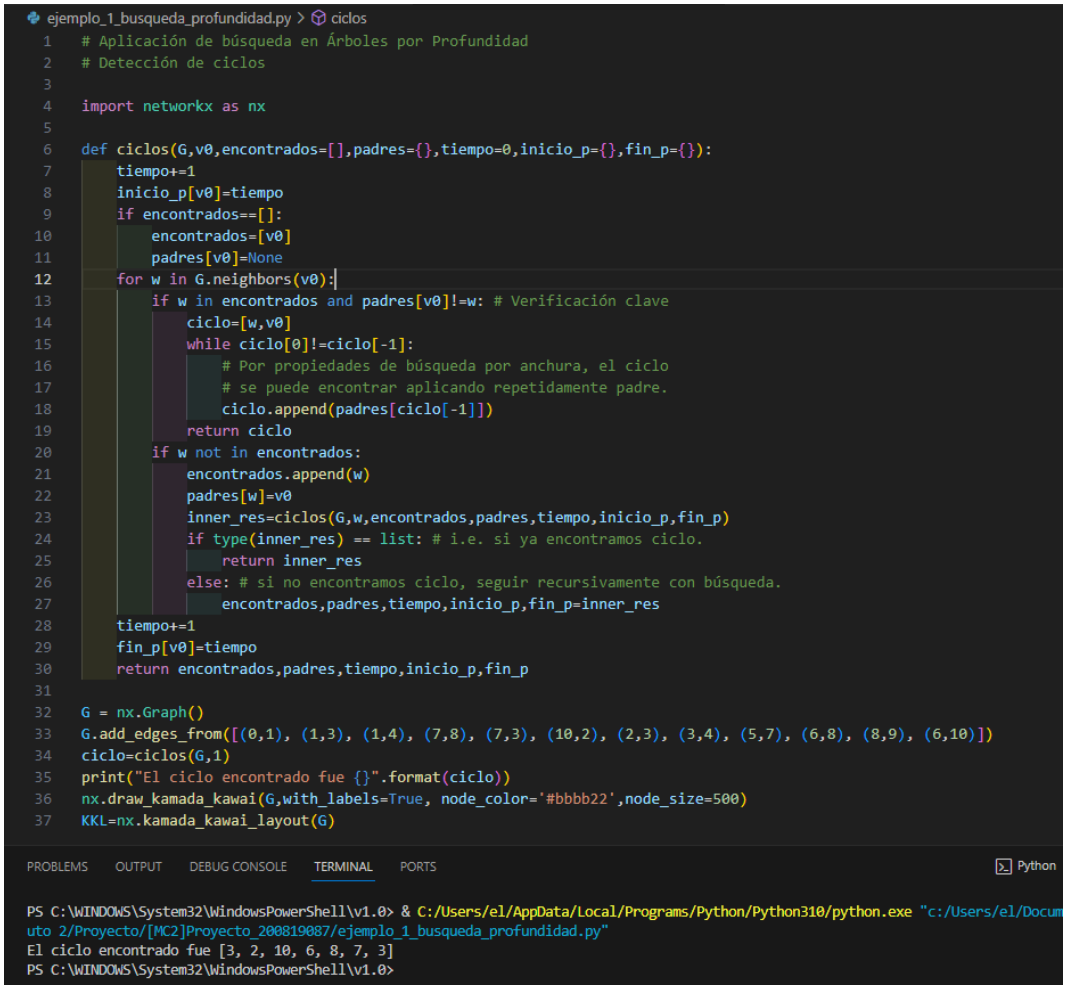
Ejemplo de Aplicación de Búsqueda en Árboles por Profundidad.

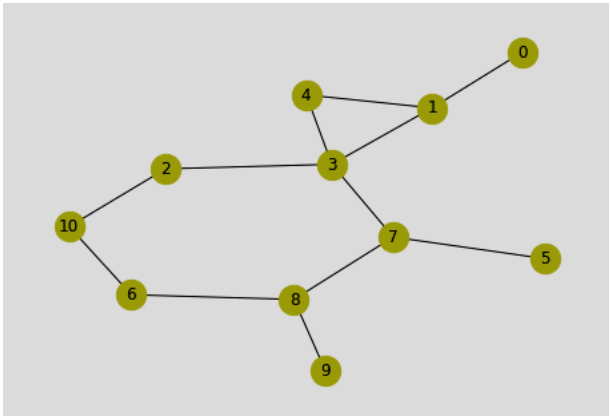
Detección de ciclos

Una búsqueda por profundidad nos permite detectar ciclos. Como vimos en la entrada anterior, todas las aristas de una gráfica conexa ***G*** son entre un vértice y uno de sus ancestros en el árbol por profundidad ***T***.

Así, una forma de detectar algorítmicamente si una gráfica tiene ciclos o no es efectuar una búsqueda por profundidad. Si todas las aristas de ***G*** son del árbol por profundidad, entonces ***G*** es un árbol y por lo tanto no tiene ciclos. Sin embargo, si en algún momento exploramos una arista que nos lleve de un vértice ***u*** a un ancestro ***v***, entonces esta cierra un ciclo y por lo tanto la gráfica no sería acíclica.

En la siguiente implementación nos bastará con encontrar un ciclo para terminar anticipadamente la ejecución.




0
4
1
2
3
10
7
5
6
8
9

Investigar un algoritmo para arboles de notación polaca (este deberá realizarlo en Júpiter y agregar capturas de pantalla de la solución con el código al correrlo con 3 notaciones)

notacion_polaca_2.ipynb > # Definición de la clase Nodo para representar nodos del árbol

+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...

```python
# Definición de la clase Nodo para representar nodos del árbol
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None

# Función para construir un árbol a partir de una expresión en notación polaca inversa
def construir_arbol_notacion_polaca(expresion):
    pila = []  # Una pila para ayudar a construir el árbol
    tokens = expresion.split()  # Dividir la expresión en tokens

    for token in tokens:
        if token.isnumeric():  # Si el token es un número, creemos un nodo y lo apilamos
            pila.append(Nodo(int(token)))
        else:
            # Si el token es un operador, creamos un nodo y lo conectamos a los dos nodos superiores en la pila
            nodo = Nodo(token)
            nodo.derecha = pila.pop()
            nodo.izquierda = pila.pop()
            pila.append(nodo)

    return pila[0]  # El nodo en la parte superior de la pila es la raíz del árbol

# Función para imprimir el árbol en orden (izquierda, raíz, derecha)
def imprimir_arbol(arbol):
    if arbol:
        imprimir_arbol(arbol.izquierda)
        print(arbol.valor, end=" ")  # Imprimimos el valor del nodo
        imprimir_arbol(arbol.derecha)

# Ejemplo 1: Notación Polaca Inversa (3 4 + 2 *)
expresion_polaca1 = "3 4 + 2 *"
arbol1 = construir_arbol_notacion_polaca(expresion_polaca1)
print("Resultado del ejemplo 1:", end=" ")
imprimir_arbol(arbol1)

# Ejemplo 2: Notación Polaca (5 1 2 + 4 * + 3 -)
expresion_polaca2 = "5 1 2 + 4 * + 3 -"
arbol2 = construir_arbol_notacion_polaca(expresion_polaca2)
print("\nResultado del ejemplo 2:", end=" ")
imprimir_arbol(arbol2)

# Ejemplo 3: Notación Polaca (7 5 3 * -)
expresion_polaca3 = "7 5 3 * -"
arbol3 = construir_arbol_notacion_polaca(expresion_polaca3)
print("\nResultado del ejemplo 3:", end=" ")
imprimir_arbol(arbol3)
```

[3] ✓ 0.0s

```
Resultado del ejemplo 1: 3 + 4 * 2
Resultado del ejemplo 2: 5 + 1 + 2 * 4 - 3
Resultado del ejemplo 3: 7 - 5 * 3
```

Conclusiones

1. La búsqueda en árboles por anchura (Breadth First Search) visita todos los nodos de un nivel antes de pasar al siguiente nivel, utilizando una cola como estructura de datos auxiliar.

2. La búsqueda en árboles por profundidad (Depth First Search) explora un camino específico de un nodo hasta encontrar una hoja o un nodo sin caminos no explorados, utilizando una pila como estructura de datos auxiliar.

3. La búsqueda en anchura tiende a encontrar soluciones más cercanas a la raíz, mientras que la búsqueda en profundidad busca soluciones en caminos más profundos del árbol.

4. Estos algoritmos tienen aplicaciones en la búsqueda de información, cálculo de rutas, juegos, y son esenciales en la resolución de problemas en campos como informática, biología, medicina e ingeniería, entre otros.

5. La elección entre la búsqueda en árboles por ancho y profundidad depende en gran medida de la ubicación de la solución buscada en el árbol. La elección del algoritmo debe adaptarse a la naturaleza del problema y la ubicación de la solución en el árbol.

Bibliografía

- Meta. (2011, 16 de septiembre). Búsqueda en amplitud y profundidad en árboles. vidasConcurrentes. https://blog.vidasconcurrentes.com/programacion/busqueda-en-profundidad-y-busqueda-en-anchura/

- Martínez Sandoval, L. I. (2022). Aplicaciones de búsqueda por profundidad. nekomath.com. https://madi.nekomath.com/P5/AplicacionesDFS.html

- Martínez Sandoval, L. I. (2022). Aplicaciones de búsqueda por anchura. nekomath.com. https://madi.nekomath.com/P5/AplicacionesBFS.html

- Prinz, P., & Stubblebine, T. (2015). Algorithms in a Nutshell, Second Edition. O'Reilly Media.