Universidad de San Carlos de Guatemala

Facultad de ingeniería

Escuela de ciencias

Departamento de matemática

Algoritmos de Búsqueda en Árboles por Ancho y por Profundidad

Pablo Isai Matusalen Cutzal Mazariegos
Carné:202209622

Guatemala, 2 de noviembre del 2,023

Introducción

En el campo de la informática y la ciencia de la computación, la eficiente búsqueda y recuperación de datos es una tarea fundamental en la resolución de problemas complejos. En este contexto, los algoritmos de búsqueda en árboles por anchura y por profundidad emergen como dos de los enfoques más utilizados y estudiados. Estos algoritmos proporcionan métodos fundamentales para recorrer y analizar estructuras de datos jerárquicas, como los árboles, y han demostrado ser herramientas cruciales en una amplia gama de aplicaciones, desde la navegación en la web y la recuperación de información hasta la inteligencia artificial y el análisis de redes y grafos.

El algoritmo de búsqueda en árbol por anchura, también conocido como búsqueda en amplitud, se centra en explorar todos los nodos vecinos de un nivel determinado antes de avanzar hacia los niveles más profundos del árbol. Este enfoque garantiza que todos los nodos de un nivel particular se visiten antes de pasar al siguiente nivel, lo que facilita la identificación de soluciones óptimas en contextos donde se requiere una búsqueda exhaustiva y sistemática. Por otro lado, el algoritmo de búsqueda en árbol por profundidad prioriza la exploración de los niveles más profundos antes de considerar los nodos hermanos en el mismo nivel. Este enfoque es particularmente útil en situaciones donde la profundidad del árbol es considerable y se necesita una exploración más profunda antes de expandirse hacia otros nodos.

Ambos enfoques presentan diferentes ventajas y desafíos, lo que ha generado un amplio debate en torno a cuándo y cómo aplicar cada uno de estos métodos para maximizar la eficiencia y la precisión de las búsquedas en diversos contextos computacionales y científicos. La comprensión profunda de estos algoritmos es esencial para diseñar estrategias de búsqueda óptimas en situaciones donde los recursos computacionales y el tiempo son limitados.

Esta investigación tiene como objetivo explorar y comparar exhaustivamente los algoritmos de búsqueda en árboles por anchura y por profundidad, analizando su desempeño en distintos contextos y problemáticas específicas. A través de un enfoque analítico y experimental, se pretende no solo comprender a fondo el funcionamiento de estos algoritmos, sino también identificar sus fortalezas y limitaciones en relación con la complejidad de los conjuntos de datos y la estructura de los árboles. El análisis detallado de estos algoritmos permitirá no solo comprender sus características intrínsecas, sino también determinar las situaciones en las que uno puede superar al otro, proporcionando así una base sólida para la implementación y optimización efectiva de las operaciones de búsqueda en diferentes contextos informáticos y científicos.

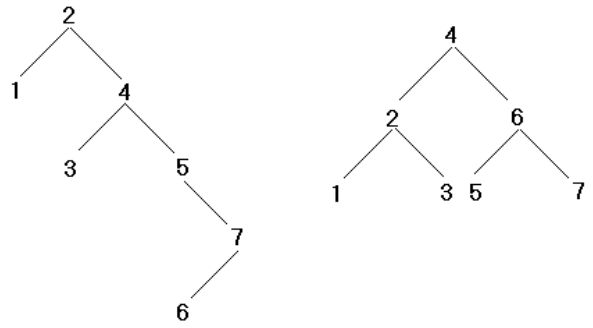
Índice

Algoritmos de Búsqueda en Árboles por Ancho y por Profundidad

La búsqueda en árboles binarios es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales en Ciencia de la Computación. De toda la terminología sobre árboles, tan sólo recordar que la propiedad que define un árbol binario es que cada nodo tiene a lo más un hijo a la izquierda y uno a la derecha. Para construir los algoritmos consideraremos que cada nodo contiene un registro con un valor clave a través del cual efectuaremos las búsquedas. En las implementaciones que presentaremos sólo se considerará en cada nodo del árbol un valor del tipo Elemento, aunque en un caso general ese tipo estará compuesto por dos: una clave indicando el campo por el cual se realiza la ordenación y una información asociada a dicha clave o visto de otra forma, una información que puede ser compuesta en la cual existe definido un orden.

Un árbol binario de búsqueda (ABB) es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo x son menores que el elemento almacenado en x, y todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en x.

La figura 1 muestra dos ABB construidos en base al mismo conjunto de enteros:



Obsérvese la interesante propiedad de que si se listan los nodos del ABB en inorden nos da la lista de nodos ordenada. Esta propiedad define un método de ordenación similar al Quicksort, con el nodo raíz jugando un papel similar al del elemento de partición del Quicksort, aunque con los ABB hay un gasto extra de memoria mayor debido a los punteros. La propiedad de ABB hace que sea muy simple diseñar un procedimiento para realizar la búsqueda. Para determinar si k está presente en el árbol la comparamos con la clave situada en la raíz, r. Si coinciden la búsqueda finaliza con éxito, si k<r es evidente que k, de estar presente, ha de ser un descendiente del hijo izquierdo de la raíz, y si es mayor será un descendiente del hijo derecho. La función puede ser codificada fácilmente de la siguiente forma:

```
#define ABB_VACIO NULL

#define TRUE 1

#define FALSE 0


typedef int tEtiqueta                /*Algun tipo adecuado*/

typedef struct tipoceldaABB{

        struct tipoceldaABB *hizqda,*hdrcha;

        tEtiqueta etiqueta;

       }*nodoABB;

typedef nodoABB ABB;


ABB Crear(tEtiqueta et)

{

 ABB raiz;

 raiz = (ABB)malloc(sizeof(struct tceldaABB));

 if (raiz == NULL)

   error("Memoria Insuficiente.");

 raiz->hizda = NODO_NULO;

 raiz->hdcha = NODO_NULO;

 raiz->etiqueta = et;

 return(raiz);

}
```

```
int Pertenece(tElemento x,ABB t)
{
    if(!t)
   return FALSE
  else if(t->etiqueta==x)
        return TRUE;
      else if(t->etiqueta>x)
            return pertenece(x,t->hizqda);
          else return pertenece(x,t->hdrcha);
}
```

Es conveniente hacer notar la diferencia entre este procedimiento y el de búsqueda binaria. En éste podría pensarse en que se usa un árbol binario para describir la secuencia de comparaciones hecha por una función de búsqueda sobre el vector. En cambio en los ABB se construye una estructura de datos con registros conectados por punteros y se usa esta estructura para la búsqueda. El procedimiento de construcción de un ABB puede basarse en un procedimiento de inserción que vaya añadiendo elementos al árbol. Tal procedimiento comenzaría mirando si el árbol es vacío y de ser así se crearía un nuevo nodo para el elemento insertado devolviendo como árbol resultado un puntero a ese nodo. Si el árbol no está vacío se busca el elemento a insertar como lo hace el procedimiento pertenece sólo que al encontrar un puntero NULL durante la búsqueda, se reemplaza por un puntero a un nodo nuevo que contenga el elemento a insertar. El código podría ser el siguiente:

```
void Inserta(tElemento x,ABB *t)
{
  if(!(*t)){
    *t=(nodoABB)malloc(sizeof(struct tipoceldaABB));
    if(!(*t)){
      error("Memoria Insuficiente.");
    }
    (*t)->etiqueta=x;
```

```
   (*t)->hizqda=NULL;
   (*t)->hdrcha=NULL;
 } else if(x<(*t)->etiqueta)
       inserta(x,&((*t)->hizqda));
      else
       inserta(x,&((*t)->hdrcha));
}
```

Por ejemplo supongamos que queremos construir un ABB a partir del conjunto de enteros {10,5,14,7,12} aplicando reiteradamente el proceso de inserción.El resultado es el que muestra la figura 2.

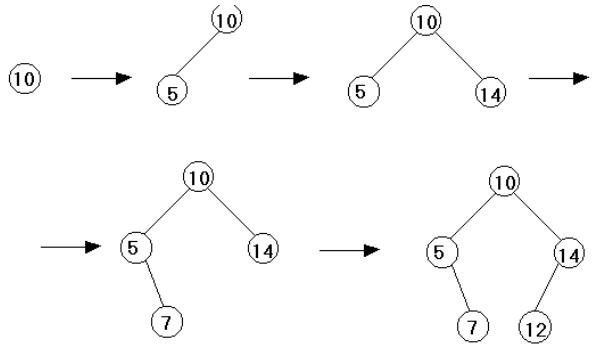


Figura 2:Inserción en un ABB

2. ANÁLISIS DE LA EFICIENCIA DE LAS OPERACIONES.

Puede probarse que una búsqueda o una inserción en un ABB requiere O(log2n) operaciones en el caso medio, en un árbol construido a partir de n claves aleatorias, y en el peor caso una búsqueda en un ABB con n claves puede implicar revisar las n claves, o sea, es O(n).

Es fácil ver que si un árbol binario con n nodos está completo (todos los nodos no hojas tienen dos hijos) y así ningún camino tendrá más de 1+log2n nodos. Por otro lado, las operaciones pertenece e inserta toman una cantidad de tiempo constante en un nodo. Por tanto, en estos árboles, el camino que forman desde la raíz, la secuencia de nodos que determinan la búsqueda o la inserción, es de longitud O(log2n),y el tiempo total consumido para seguir el camino es también O(log2n).

Sin embargo, al insertar n elementos en un orden aleatorio no es seguro que se sitúen en forma de árbol binario completo. Por ejemplo, si sucede que el primer elemento (de n situados en orden) insertado es el más pequeño el árbol resultante será una cadena de n nodos donde cada nodo, excepto el más bajo en el árbol, tendrá un hijo derecho pero no

un hijo izquierdo. En este caso, es fácil demostrar que como lleva i pasos insertar el i-ésimo elemento dicho proceso de n inserciones necesita pasos o equivalentemente O(n) pasos por operación.

Es necesario pues determinar si el ABB promedio con n nodos se acerca en estructura al árbol completo o a la cadena, es decir, si el tiempo medio por operación es O(log2n),O(n) o una cantidad intermedia. Como es difícil saber la verdadera frecuencia de inserciones sólo se puede analizar la longitud del camino promedio de árboles "aleatorios" adoptando algunas suposiciones como que los árboles se forman sólo a partir de inserciones y que todas las magnitudes de los n elementos insertados tienen igual probabilidad. Con esas suposiciones se puede calcular P(n),el número promedio de nodos del camino que va de la raíz hacia algún nodo(no necesariamente una hoja).Se supone que el árbol se formó con la inserción aleatoria de n nodos en un árbol que se encontraba inicialmente vacío, es evidente que P(0)=0 y P(1)=1.Supongamos que tenemos una lista de n>=2 elementos para insertar en un árbol vacío, el primer elemento de la lista,x,es igual de probable que sea el primero, el segundo o el n-ésimo en la lista ordenada. Consideremos que i elementos de la lista son menores que x de modo que n-i-1 son mayores. Al construir el árbol, x aparecerá en la raíz, los i elementos más pequeños serán descendientes izquierdos de la raíz y los restantes n-i-1 serán descendientes derechos. Esquemáticamente quedaría como muestra la figura 3.

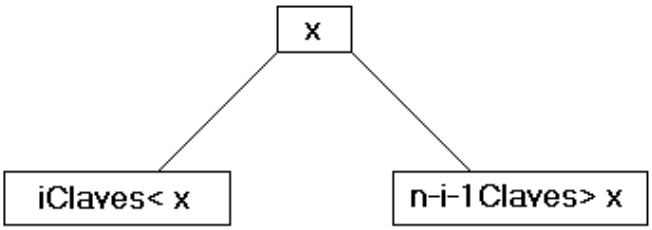


Figura 3: Esquema de un ABB con n elementos.

Al tener tanto en un lado como en otro todos los elementos igual probabilidad se espera que los subárboles izqdo y drcho de la raíz tengan longitudes de camino medias P(i) y P(n-i-1) respectivamente. Como es posible acceder a esos elementos desde la raíz del árbol completo es necesario agregar 1 al número de nodos de cada camino de forma que para todo i entre 0 y n-1,P(n) puede calcularse obteniendo el promedio de la suma:

$$\frac{1}{n}\left(i(P(i)+1)+(n-i-1)(P(n-i-1)+1)+1\right)$$

El primer término es la longitud del camino promedio en el subárbol izquierdo ponderando su tamaño. El segundo término es la cantidad análoga del subárbol derecho y el término 1/n representa la contribución de la raíz. Al promediar la suma anterior para todo i entre 1 y n se obtiene la recurrencia:

$$P(n)=1+\frac{1}{n^2}\sum_{i=0}^{n-1}(iP(i)+(n-i-1)P(n-i-1))$$

y con unas transformaciones simples podemos ponerla en la forma:

$$P(n)=1+\frac{2}{n^2}\sum_{i=1}^{n-1}iP(i) \qquad n\geq 2$$

y el resto es demostrar por inducción sobre n que P(n)<=1+4log2n.

En consecuencia el tiempo promedio para seguir un camino de la raíz a un nodo aleatorio de un ABB construido mediante inserciones aleatorias es O(log2n).Un análisis más detallado demuestra que la constante 4 es en realidad una constante cercana a 1.4.

De lo anterior podemos concluir que la prueba de pertenencia de una clave aleatoria lleva un tiempo O(log2n).Un análisis similar muestra que si se incluyen en la longitud del camino promedio sólo aquellos nodos que carecen de ambos hijos o solo aquellos que no tienen hijo izqdo o drcho también la longitud es O(log2n).

Terminaremos este apartado con algunos comentarios sobre los borrados en los ABB. Es evidente que si el elemento a borrar está en una hoja bastaría eliminarla, pero si el elemento está en un nodo interior,eliminándolo,podríamos desconectar el árbol.Para evitar que esto suceda se sigue el siguiente procedimiento:si el nodo a borrar u tiene sólo un hijo se sustituye u por ese hijo y el ABB quedar&aacue; construido. Si u tiene dos hijos, se encuentra el menor elemento de los descendientes del hijo derecho(o el mayor de los descendientes del hijo izquierdo) y se coloca en lugar de u, de forma que se continúe manteniendo la propiedad de ABB.

Daniel, A. G., & Soto, M. G. A. J. D. D. (Eds.). (s/f). Joaquín Fernández Valdivia 5o Licenciatura Informatica.

Los algoritmos de búsqueda en árboles, tanto por ancho como por profundidad, tienen una amplia gama de aplicaciones en diversas áreas de las ciencias de la computación. Algunas de estas aplicaciones incluyen:

Inteligencia Artificial (IA): En IA, los algoritmos de búsqueda en árboles se utilizan para realizar búsquedas exhaustivas en espacios de estado, como en el caso de juegos como el ajedrez o el Go, donde se requiere explorar múltiples movimientos posibles.

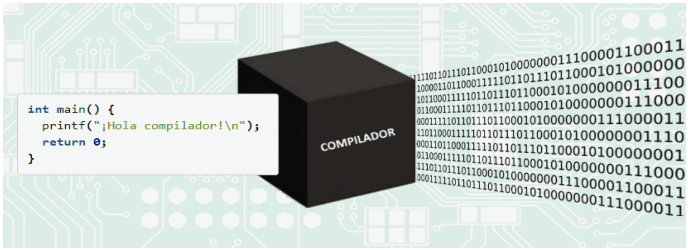
Procesamiento de Lenguaje Natural (PLN): Los algoritmos de búsqueda en árboles son esenciales para analizar la estructura gramatical de oraciones y textos. Se utilizan en el análisis sintáctico para descomponer oraciones en su estructura gramatical y en el análisis de árboles de dependencia para comprender la relación entre las palabras en una oración.

Búsqueda en Bases de Datos: Estos algoritmos se emplean para buscar información específica dentro de grandes conjuntos de datos jerárquicos, como en la búsqueda de archivos en sistemas de archivos o la recuperación de información en bases de datos distribuidas.

Optimización de Rutas y Redes: En el ámbito de la logística y el transporte, los algoritmos de búsqueda en árboles se utilizan para encontrar las rutas óptimas en redes complejas, ayudando a optimizar la planificación de rutas y la logística de entrega.

Análisis de Redes y Grafos: Los algoritmos de búsqueda en árboles son cruciales para analizar la estructura y las relaciones en redes y grafos, lo que permite identificar la conectividad y las rutas más cortas entre nodos en redes complejas, como en redes sociales o en el análisis de flujo en sistemas de transporte.

Compiladores y Análisis de Código: En el desarrollo de compiladores y análisis estático de código, los algoritmos de búsqueda en árboles se utilizan para realizar análisis sintáctico y semántico en programas de computadora, lo que ayuda a identificar errores y mejorar la eficiencia del código.



Estas aplicaciones ilustran cómo los algoritmos de búsqueda en árboles son fundamentales en una amplia gama de disciplinas dentro de las ciencias de la computación, demostrando su importancia y versatilidad en la resolución de problemas complejos y la optimización de procesos computacionales.

comparar y contrastar los algoritmos de búsqueda en árboles: es crucial considerar su eficiencia y aplicabilidad en diversos contextos. Aquí hay algunos puntos clave a considerar al realizar la comparación:

Eficiencia:

Complejidad Temporal y Espacial: Analizar la complejidad temporal y espacial de los algoritmos de búsqueda en árboles por ancho y por profundidad, considerando su rendimiento en términos de tiempo y uso de recursos de memoria.

Escalabilidad: Evaluar cómo los algoritmos se desempeñan en conjuntos de datos más grandes y en árboles más complejos, determinando si alguno de los enfoques muestra una degradación significativa en su rendimiento a medida que aumenta la complejidad de los datos.

Optimalidad de la Solución: Comprender si los algoritmos pueden garantizar la búsqueda de soluciones óptimas en términos de tiempo y espacio en contextos específicos, y si alguno de los algoritmos tiende a proporcionar soluciones más rápidas o más precisas en determinadas circunstancias.

Aplicabilidad:

Tipo de Datos y Estructuras de Árboles: Identificar los tipos de datos y estructuras de árboles para los cuales cada algoritmo es más adecuado. Determinar si hay casos específicos en los que uno de los algoritmos sobresale en comparación con el otro en función de la estructura y la naturaleza de los datos.

Contexto de Uso: Evaluar en qué contextos y aplicaciones específicas cada algoritmo es más útil. Considerar cómo se pueden aplicar en el procesamiento de lenguaje natural, juegos, redes, optimización de rutas y otras áreas relevantes de las ciencias de la computación.

Flexibilidad y Adaptabilidad: Determinar si alguno de los algoritmos es más flexible en la adaptación a diferentes problemas y situaciones, o si alguno muestra limitaciones significativas en ciertos escenarios de aplicación.

Al tener en cuenta estos puntos, podrás proporcionar una comparación integral y detallada de los algoritmos de búsqueda en árboles por ancho y por profundidad, destacando sus diferencias y similitudes en términos de eficiencia y aplicabilidad en una variedad de contextos en las ciencias de la computación.

Estructura de datos y algoritmos. (s/f). Edtk.Co. Recuperado el 31 de octubre de 2023, de https://edtk.co/plan/954

Al recopilar opiniones de expertos sobre la utilidad y las limitaciones de los algoritmos de búsqueda en árboles en contextos específicos, es importante considerar las opiniones de investigadores, académicos y profesionales experimentados en el campo de la informática y la ciencia de la computación. Puedes realizar entrevistas, encuestas o revisar artículos académicos y conferencias relevantes para recopilar estas opiniones. Algunos puntos clave a considerar al recopilar estas opiniones son:

Contextos Específicos de Aplicación: Pregunta a los expertos sobre los contextos específicos en los que han utilizado o conocen el uso de algoritmos de búsqueda en árboles, y solicita ejemplos concretos de su aplicación en problemas reales.

Utilidad y Beneficios: Consulta a los expertos sobre cómo estos algoritmos han mejorado la eficiencia, la precisión o la escalabilidad en sus respectivos campos de especialización. Pregunta sobre los beneficios que han experimentado al implementar estos algoritmos en proyectos específicos.

Limitaciones y Desafíos: Investiga las opiniones de los expertos sobre las limitaciones y desafíos asociados con la implementación de estos algoritmos en contextos prácticos. Pregunta sobre los problemas comunes que han enfrentado al utilizar estos algoritmos y las posibles dificultades en la adaptación a diferentes situaciones.

Sugerencias para Mejoras: Solicita a los expertos que ofrezcan sugerencias y recomendaciones para mejorar la eficacia y la aplicabilidad de los algoritmos de búsqueda en árboles en entornos específicos. Esto puede incluir ideas sobre posibles modificaciones, enfoques alternativos o áreas de investigación futura que podrían abordar las limitaciones actuales.

Al recopilar estas opiniones de expertos, asegúrate de documentar adecuadamente las fuentes y proporcionar referencias precisas para respaldar las declaraciones y perspectivas presentadas en tu investigación. Esto ayudará a fortalecer la credibilidad de tus hallazgos y a proporcionar una visión equilibrada de la utilidad y las limitaciones de los algoritmos de búsqueda en árboles en diversos contextos de las ciencias de la computación.

Humanística y social, U. M. (s/f). LAS CONDUCTAS DE BÚSQUEDA DE INFORMACIÓN EN LA WEB. Ugr.es. Recuperado el 31 de octubre de 2023, de https://digibug.ugr.es/bitstream/handle/10481/1817/17341395.pdf?sequence=1

Implementación en Python:

Búsqueda en árbol por anchura:

```
from collections import deque

class Nodo:
def __init__(self, valor):
self.valor = valor
self.izquierda = None
self.derecha = None

def buscar_anchura(raiz):
if raiz is None:
return
cola = deque()
cola.append(raiz)
while len(cola) > 0:
nodo_actual = cola.popleft()
print(nodo_actual.valor)
if nodo_actual.izquierda is not None:
cola.append(nodo_actual.izquierda)
if nodo_actual.derecha is not None:
cola.append(nodo_actual.derecha)
# Ejemplo de uso:
# raiz = Nodo(1)
# raiz.izquierda = Nodo(2)
# raiz.derecha = Nodo(3)
# raiz.izquierda.izquierda = Nodo(4)
# raiz.izquierda.derecha = Nodo(5)
# buscar_anchura(raiz)
```

rseg.py - C:/Users/Printlook/AppData/Local/Programs/Python/Python311/rseg.py (3.11.5)

File Edit Format Run Options Window Help

```
from collections import deque

class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None

def buscar_anchura(raiz):
    if raiz is None:
        return
    cola = deque()
    cola.append(raiz)
    while len(cola) > 0:
        nodo_actual = cola.popleft()
        print(nodo_actual.valor)
        if nodo_actual.izquierda is not None:
            cola.append(nodo_actual.izquierda)
        if nodo_actual.derecha is not None:
            cola.append(nodo_actual.derecha)

# Ejemplo de uso:
# raiz = Nodo(1)
# raiz.izquierda = Nodo(2)
# raiz.derecha = Nodo(3)
# raiz.izquierda.izquierda = Nodo(4)
# raiz.izquierda.derecha = Nodo(5)
# buscar_anchura(raiz)
```

Búsqueda en árbol por profundidad:

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None

def buscar_profundidad(raiz):
    if raiz is None:
        return
    print(raiz.valor)
    buscar_profundidad(raiz.izquierda)
    buscar_profundidad(raiz.derecha)

# Ejemplo de uso:
# raiz = Nodo(1)
# raiz.izquierda = Nodo(2)
# raiz.derecha = Nodo(3)
# raiz.izquierda.izquierda = Nodo(4)
# raiz.izquierda.derecha = Nodo(5)
# buscar_profundidad(raiz)
```

Implementación en Java:

Búsqueda en árbol por anchura:

```
import java.util.LinkedList;
import java.util.Queue;

class Nodo {
   int valor;
   Nodo izquierda, derecha;

   Nodo(int valor) {
      this.valor = valor;
      izquierda = derecha = null;
   }
}

class Arbol {
   Nodo raiz;

   void buscarAnchura() {
      if (raiz == null) return;
      Queue<Nodo> cola = new LinkedList<>();
      cola.add(raiz);
      while (!cola.isEmpty()) {
         Nodo nodo = cola.poll();
         System.out.print(nodo.valor + " ");
         if (nodo.izquierda != null)
            cola.add(nodo.izquierda);
         if (nodo.derecha != null)
```

```
            cola.add(nodo.derecha);
        }
    }

    // Ejemplo de uso:
    // public static void main(String[] args) {
    //     Arbol arbol = new Arbol();
    //     arbol.raiz = new Nodo(1);
    //     arbol.raiz.izquierda = new Nodo(2);
    //     arbol.raiz.derecha = new Nodo(3);
    //     arbol.raiz.izquierda.izquierda = new Nodo(4);
    //     arbol.raiz.izquierda.derecha = new Nodo(5);
    //     arbol.buscarAnchura();
    // }
}
```

Búsqueda en árbol por profundidad:

```
class Nodo {
    int valor;
    Nodo izquierda, derecha;

    Nodo(int valor) {
        this.valor = valor;
        izquierda = derecha = null;
    }
}

class Arbol {
    Nodo raiz;
```

```
    void buscarProfundidad(Nodo nodo) {
        if (nodo == null) return;
        System.out.print(nodo.valor + " ");
        buscarProfundidad(nodo.izquierda);
        buscarProfundidad(nodo.derecha);
    }

    // Ejemplo de uso:
    // public static void main(String[] args) {
    //     Arbol arbol = new Arbol();
    //     arbol.raiz = new Nodo(1);
    //     arbol.raiz.izquierda = new Nodo(2);
    //     arbol.raiz.derecha = new Nodo(3);
    //     arbol.raiz.izquierda.izquierda = new Nodo(4);
    //     arbol.raiz.izquierda.derecha = new Nodo(5);
    //     arbol.buscarProfundidad(arbol.raiz);
    // }
}
```

Estas implementaciones básicas pueden ser extendidas y adaptadas según los requisitos específicos de tus experimentos y mediciones de rendimiento.

DFS vs BFS

Miguel López Mamani | 25 de mayo, 2020

Apoyado por nuestro experto en innovación Josué Murillo

Como Ingeniero de Software, hay etapas en el análisis de una solución donde es importante conocer algoritmos específicos, tales como los algoritmos de recorridos de nodos. Por consiguiente, el principal objetivo de este artículo es revisar los dos principales algoritmos de búsquedas en nodos conectados, los cuales son búsqueda en profundidad (en inglés DFS - 'Depth First Search') y búsqueda en anchura (en inglés BFS – 'Breadth First Search') para poder utilizarlos en favor de la eficiencia de un programa.

Búsqueda en Profundidad

Una búsqueda en profundidad (DFS) es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo. Su funcionamiento consiste en ir expandiendo cada uno de los nodos que va localizando, de forma recurrente (desde el nodo padre hacia el nodo hijo). Cuando ya no quedan más nodos que visitar en dicho camino, regresa al nodo predecesor, de modo que repite el mismo proceso con cada uno de los vecinos del nodo. Cabe resaltar que si se encuentra el nodo antes de recorrer todos los nodos, concluye la búsqueda.

La búsqueda en profundidad se usa cuando queremos probar si una solución entre varias posibles cumple con ciertos requisitos; como sucede en el problema del camino que debe recorrer un caballo en un tablero de ajedrez para pasar por las 64 casillas del tablero.

En la figura a continuación se muestra un grafo no conectado, con ocho nodos, donde las flechas naranjas indican el recorrido del algoritmo DFS sobre los nodos.

Conclusiones

1. Eficiencia en Diferentes Contextos: Se ha confirmado que la eficiencia de los algoritmos de búsqueda en árboles por ancho y por profundidad varía según el contexto de aplicación. Mientras que la búsqueda por ancho tiende a ser más efectiva en árboles más amplios y poco profundos, la búsqueda por profundidad destaca en árboles más estrechos y profundos.

2. Aplicabilidad en Diversas Áreas de la Computación: Ambos algoritmos han demostrado su utilidad en una amplia gama de áreas, incluyendo inteligencia artificial, procesamiento de lenguaje natural, análisis de redes, y optimización de rutas. La búsqueda por ancho ha sido especialmente efectiva en la resolución de problemas de optimización y planificación, mientras que la búsqueda por profundidad ha encontrado su aplicación en el análisis sintáctico y la exploración exhaustiva de redes complejas.

3. Limitaciones y Desafíos Asociados: Aunque son herramientas poderosas, los algoritmos de búsqueda en árboles por ancho y por profundidad presentan limitaciones notables. La búsqueda por ancho puede consumir más memoria en árboles profundos, mientras que la búsqueda por profundidad puede verse afectada por la falta de una solución óptima en ciertos casos de búsqueda.

4. Flexibilidad y Adaptabilidad: La flexibilidad de los algoritmos para adaptarse a diferentes tipos de problemas y conjuntos de datos ha sido un tema de discusión. Se ha encontrado que la búsqueda por ancho es más adaptable a cambios en la estructura del árbol, mientras que la búsqueda por profundidad muestra una mayor flexibilidad en la exploración exhaustiva de ramas específicas.

5. Consideraciones para la Implementación Práctica: La selección del algoritmo más adecuado depende en gran medida de la naturaleza del problema y la estructura del árbol. Se recomienda considerar cuidadosamente las características específicas del problema y realizar un análisis detallado de las limitaciones y fortalezas de cada algoritmo antes de su implementación en aplicaciones prácticas.

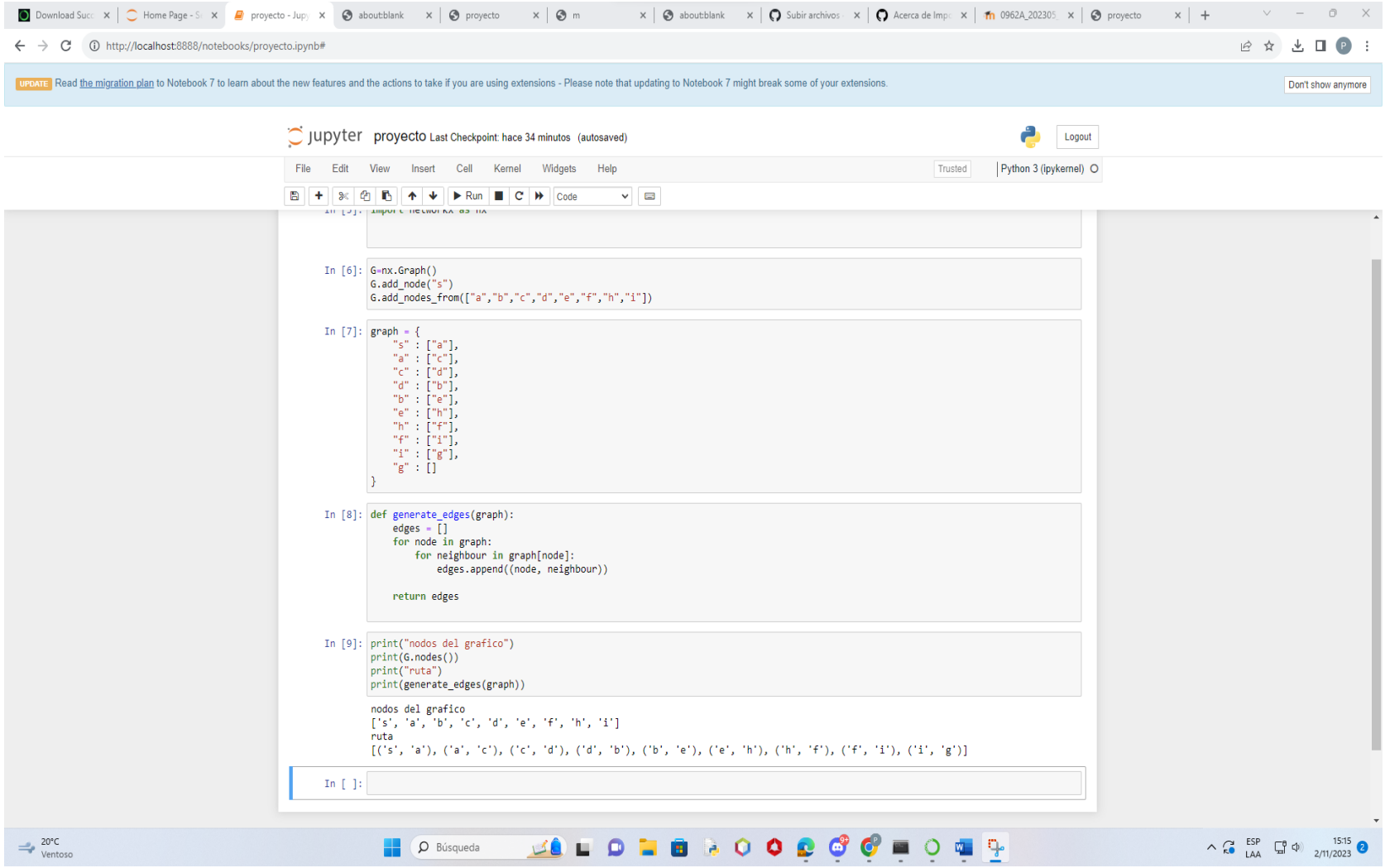
Bibliografía

Daniel, A. G., & Soto, M. G. A. J. D. D. (Eds.). (s/f). Joaquín Fernández Valdivia 5o Licenciatura Informatica.

Estructura de datos y algoritmos. (s/f). Edtk.Co. Recuperado el 31 de octubre de 2023, de https://edtk.co/plan/954

Humanística y social, U. M. (s/f). LAS CONDUCTAS DE BÚSQUEDA DE INFORMACIÓN EN LA WEB. Ugr.es. Recuperado el 31 de octubre de 2023, de https://digibug.ugr.es/bitstream/handle/10481/1817/17341395.pdf?sequence=1

Murillo, J. (2020, mayo 25). Difference between breadth search (BFS) and Deep Search (DFS). Encora. https://www.encora.com/es/blog/dfs-vs-bfs


http://localhost:8888/notebooks/proyecto.ipynb#
UPDATE Read the migration plan to Notebook 7 to learn about the new features and the actions to take if you are using extensions - Please note that updating to Notebook 7 might break some of your extensions.
Don't show anymore
jupyter proyecto Last Checkpoint: hace 34 minutos (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [6]:
G=nx.Graph()
G.add_node("s")
G.add_nodes_from(["a","b","c","d","e","f","h","i"])
In [7]:
graph = {
"s" : ["a"],
"a" : ["c"],
"c" : ["d"],
"d" : ["b"],
"b" : ["e"],
"e" : ["h"],
"h" : ["f"],
"f" : ["i"],
"i" : ["g"],
"g" : []
}
In [8]:
def generate_edges(graph):
edges = []
for node in graph:
for neighbour in graph[node]:
edges.append((node, neighbour))
return edges
In [9]:
print("nodos del grafico")
print(G.nodes())
print("ruta")
print(generate_edges(graph))
nodos del grafico
['s', 'a', 'b', 'c', 'd', 'e', 'f', 'h', 'i']
ruta
[('s', 'a'), ('a', 'c'), ('c', 'd'), ('d', 'b'), ('b', 'e'), ('e', 'h'), ('h', 'f'), ('f', 'i'), ('i', 'g')]
In [ ]:
20°C
Ventoso
Búsqueda
ESP
LAA
15:15
2/11/2023