			Fecha: **02/11/2023**
Universidad de San Carlos de Guatemala	Matemática para Computación 2 Sección A		
Facultad de ingeniería	Ing. José Alfredo González Díaz		
Escuela de ciencias	Aux. Edgar Daniel Cil Peñate		
Departamento de matemática	Segundo semestre 2023		
Nombre: **Héctor Daniel Ortiz Osorio**		Carné:	**202203806**

Proyecto

Proyecto de Investigación: Algoritmos de Búsqueda en Árboles por Ancho y por Profundidad.

1. Investigación sobre Algoritmos de Búsqueda en Árboles.

Búsqueda en Profundidad

"La búsqueda en profundidad, conocida como DFS (por sus siglas en inglés, Depth-First Search), es un algoritmo utilizado para explorar grafos. Funciona expandiendo los nodos de manera recursiva, avanzando desde el nodo padre hacia sus descendientes. Cuando no hay más nodos por visitar en un camino, se retrocede al nodo predecesor y se repite el proceso con los vecinos de ese nodo. Es relevante destacar que, si se encuentra el nodo objetivo antes de explorar todos los nodos, la búsqueda se detiene.

Este enfoque es útil cuando se busca determinar si una de varias soluciones posibles cumple con ciertos requisitos. Por ejemplo, en el problema de encontrar el recorrido de un caballo en un tablero de ajedrez que debe pasar por las 64 casillas."

Búsqueda en Anchura

La búsqueda en anchura, conocida como BFS (por sus siglas en inglés, Breadth-First Search), es un algoritmo que explora los nodos de un grafo. Este proceso comienza desde el nodo raíz, el cual se elige punto de partida en el caso de un

grafo. Luego, se procede a explorar todos los nodos vecinos de este punto. Seguidamente, se repite el proceso para cada uno de los vecinos, explorando a su vez sus nodos adyacentes. Este ciclo se repite hasta que se haya recorrido todo el grafo. Es importante destacar que, si se encuentra el nodo objetivo antes de haber explorado todos los nodos, la búsqueda concluye.

La búsqueda por anchura es esencial en algoritmos donde la selección del mejor camino en cada etapa del recorrido resulta crítica.

2. **Identificación de Aplicaciones de algoritmos en las ciencias de la computación.**

Aplicaciones de DFS

1. **Búsqueda de Rutas en Mapas:**
    - Los algoritmos BFS se utilizan en aplicaciones de navegación y mapas para encontrar la ruta más corta entre dos ubicaciones. Por ejemplo, en servicios de mapas como Google Maps.
2. **Redes Sociales y Grafo de Amistades:**
    - En las redes sociales, BFS se aplica para encontrar conexiones entre usuarios. Por ejemplo, en la recomendación de amigos o en la visualización de redes de contactos.
3. **Búsqueda de Caminos en Grafos Ponderados:**
    - Se utiliza en la búsqueda de caminos mínimos en grafos ponderados. Por ejemplo, en la determinación de la ruta más eficiente en un sistema de transporte público.
4. **Web Crawling y Rastreo de Enlaces:**
    - Los motores de búsqueda utilizan BFS para rastrear y indexar páginas web, siguiendo enlaces de página a página.

Aplicaciones de BFS

1. **Resolución de Laberintos y Caminos:**
    - DFS es efectivo para encontrar soluciones en problemas de búsqueda de caminos, como la resolución de laberintos.
2. **Árboles de Decisión en Aprendizaje Automático:**
    - Se utiliza en algoritmos de aprendizaje automático para construir árboles de decisión que clasifican o predicen resultados.
3. **Backtracking y Resolución de Problemas:**
    - DFS es fundamental en técnicas de backtracking para resolver problemas como el Sudoku, N-Queens, entre otros.
4. **Análisis de Grafos y Detectar Componentes Conectados:**
    - Se aplica en la detección de ciclos, componentes conectados y puntos de articulación en grafos.
5. **Optimización de Rutas en Logística:**
    - En logística, DFS puede utilizarse para encontrar rutas eficientes de entrega o recogida de mercancías.

3. Implementación y Experimentación en los Algoritmos de Búsqueda.

EJEMPLO 1

A continuación, usaremos Jupyter, para implementar el código, dando un ejemplo básico de como se puede aplicar los algoritmos de búsqueda, utilizando diversos lenguajes de programación, se usará el Lenguaje de JavaScript.

```
class Nodo {
    constructor(valor) {
        this.valor = valor;
        this.hijos = [];
    }
}

function bfs(raiz) {
    let cola = [raiz];
    let visitados = new Set();

    while (cola.length > 0) {
        let nodo = cola.shift();
        if (!visitados.has(nodo)) {
            console.log(nodo.valor);
            visitados.add(nodo);
            cola.push(...nodo.hijos);
        }
    }
}
function dfs(nodo) {
    if (nodo !== null) {
        console.log(nodo.valor);
        nodo.hijos.forEach(hijo => dfs(hijo));
    }
}

// Creamos un arbol
const nodo1 = new Nodo(1);
const nodo2 = new Nodo(2);
const nodo3 = new Nodo(3);
const nodo4 = new Nodo(4);
const nodo5 = new Nodo(5);

nodo1.hijos = [nodo2, nodo3];
nodo2.hijos = [nodo4, nodo5];

// Prrobamos y aplicamos el BFS
console.log("Recorrido BFS:");
bfs(nodo1);

// Prrobamos y aplicamos el DFS
console.log("\nRecorrido DFS:");
dfs(nodo1);
```

Como resultado nos da lo siguiente:

```
Recorrido BFS:
1
2
3
4
5

Recorrido DFS:
1
2
4
5
3
```

En el recorrido BFS, primero visita el nodo raíz (1), luego los nodos de nivel 2 (2 y 3), y finalmente los nodos de nivel 3 (4 y 5).

En el recorrido DFS, se explora lo más profundo posible a lo largo de cada rama antes de retroceder. Por lo tanto, se visita el nodo 1, luego su primer hijo (2) y así sucesivamente hasta llegar al último nodo (5) antes de retroceder y continuar la exploración.

EJEMPLO 2

Para el segundo ejemplo se usará NetBeans, para ejecutar el código, el cual será en lenguaje de programación Java.

```java
package proyecto.mc2;

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

class Nodos {
    int data;
    Nodos left, right; // Las Referencias de los nodos izquierdo y derecho

    public Nodos(int item) { // Constructor de la clase Nodos
        data = item;
        left = right = null;
    }
}

class BinarioArbol {
    Nodos root; // Raíz del árbol

    void printLevelOrder() { // Método para realizar la búsqueda en anchura

        Queue<Nodos> queue = new LinkedList<Nodos>(); // Crear una cola para almacenar nodos
        queue.add(e: root); // Agregar la raíz a la cola
        while (!queue.isEmpty()) { // Mientras la cola no esté vacía
            Nodos tempNode = queue.poll(); // Sacar el primer nodo de la cola
            System.out.print(tempNode.data + " "); // Imprimir el valor del nodo
            if (tempNode.left != null) { // Si hay un nodo izquierdo, agregarlo a la cola
                queue.add(e: tempNode.left);
            }
            if (tempNode.right != null) { // Si hay un nodo derecho, agregarlo a la cola
                queue.add(e: tempNode.right);
```

```
31              }
32          }
33      }
34
35      void printInorder(Nodos nodos) { // Método para realizar la búsqueda en profundidad
36          if (nodos == null) { // Si el nodo es nulo, retornar
37              return;
38          }
39          printInorder( nodos: nodos.left); // Recursivamente imprimir el subárbol izquierdo
40          System.out.print(nodos.data + " "); // Imprimir el valor del nodo actual
41          printInorder( nodos: nodos.right); // Recursivamente imprimir el subárbol derecho
42      }
43  }
44
45  public class ProyectoMc2 {
46
47      public static void main(String[] args) {
48
49          BinarioArbol tree = new BinarioArbol();
50
51          tree.root = new Nodos( item: 1); // Crear la raiz del árbol con valor 1
52          tree.root.left = new Nodos( item: 2); // Crear un nodo izquierdo con valor 2
53          tree.root.right = new Nodos( item: 3); // Crear un nodo derecho con valor 3
54          tree.root.left.left = new Nodos( item: 4); // Crear un nodo izquierdo del nodo izquierdo con valor 4
55          tree.root.left.right = new Nodos( item: 5); // Crear un nodo derecho del nodo izquierdo con valor 5
56
57          System.out.println( x: "Busqueda en Arbol por ancho:"); // Imprimimos el  mensaje
58          tree.printLevelOrder(); // Llamamos al método de búsqueda en anchura
59
60          System.out.println( x: "\nBusqueda en Arbol por profundidad:"); // Imprimimos el  mensaje
61          tree.printInorder( nodos: tree.root); // Llamamos al método de búsqueda en profundidad
62      }
63  }
64
```

Como resultado nos da lo siguiente:

En la búsqueda en árbol por anchura (BFS), se visitan los nodos en el mismo nivel antes de moverse al siguiente nivel. Por lo tanto, el orden sería: 1, 2, 3, 4, 5.

En la búsqueda en árbol por profundidad (DFS), se exploran completamente las ramas antes de retroceder. En este caso, el recorrido inorden es: 4, 2, 5, 1, 3.

```
Busqueda en Arbol por ancho:
1 2 3 4 5
Busqueda en Arbol por profundidad:
4 2 5 1 3 BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Entrevistas y Encuestas.

Experto en Algoritmos de Búsqueda:

¿En qué situaciones específicas recomendaría el uso de un algoritmo de búsqueda en amplitud en lugar de un algoritmo de búsqueda en profundidad?

- "Recomendaría el uso de búsqueda en amplitud cuando se necesita encontrar la solución más corta o la ruta óptima en un grafo no ponderado. Por otro lado, la búsqueda en profundidad puede ser más eficiente en grafos grandes con profundidades significativas."

¿Qué consideraciones clave deben tenerse en cuenta al implementar un algoritmo de búsqueda en profundidad en un grafo con posibles ciclos?

- "Al implementar un algoritmo de búsqueda en profundidad, es crucial tener en cuenta la posibilidad de caer en ciclos. Se deben mantener registros de los nodos visitados para evitar ciclos infinitos y garantizar que el algoritmo termine correctamente."

Utilidad de Algoritmos de Búsqueda en Aplicaciones Prácticas:

En una escala del 1 al 5, ¿cómo evaluaría la efectividad de la búsqueda en amplitud para encontrar la ruta más corta en un sistema de navegación?

- Calificación para la efectividad de la búsqueda en amplitud: 4
- "La búsqueda en amplitud ha sido altamente efectiva en sistemas de navegación para encontrar rutas óptimas en ciudades congestionadas."

¿Ha experimentado situaciones en las que la búsqueda en profundidad fue la mejor opción para resolver un problema específico en su campo?

- "En ciencias de la computación teóricas, la búsqueda en profundidad ha sido invaluable para explorar estructuras de datos complejas y analizar relaciones entre nodos en grafos profundos."

5. Análisis y Comparación.

Eficiencia y Complejidad:

1. **Búsqueda en Amplitud (BFS)**
    - **Eficiencia:** Es altamente eficaz para encontrar la solución más corta en grafos no ponderados.
    - **Complejidad:** Tiene una complejidad de tiempo, donde V es el número de vértices y E es el número de aristas.
2. **Búsqueda en Profundidad (DFS)**
    - **Eficiencia:** Puede ser más eficiente en grafos grandes con mayor profundidad, especialmente cuando se necesitan exploraciones profundas.
    - **Complejidad:** Tiene una complejidad de tiempo, similar al de BFS, pero su uso puede variar según la configuración específica del grafo a implementar.

Aplicabilidad:

- **Búsqueda en Amplitud (BFS)**
    - Recomendado para encontrar rutas óptimas en sistemas de navegación, especialmente en ciudades congestionadas, por ejemplo, el uso de aplicaciones como Waze o Google Maps, donde la misma aplicación te da alternativas a rutas, en donde el objetivo es encontrar una ruta que se pueda llegar lo más antes posibles a su destino.
    - Útil cuando se requiere encontrar la solución más corta en grafos no ponderados.
    - Efectivo para determinar la conectividad entre nodos en redes sociales o gráficos de relaciones.
- **Búsqueda en Profundidad (DFS):**
    - Valioso en ciencias de la computación teóricas para explorar estructuras de datos complejas y analizar relaciones en grafos profundos.

- Puede ser preferible en grafos grandes con profundidades significativas, especialmente cuando se necesita explorar en profundidad.

6. Investigación Adicional

Notación Polaca

La notación polaca, también conocida como notación Łukasiewicz o notación polaca normal, es un método matemático donde los operadores se colocan antes de sus operandos. Esto difiere de la notación infija más común, donde los operadores se sitúan entre los operandos. Asimismo, se distingue de la notación polaca inversa, en la cual los operadores siguen a los operandos.

Por ejemplo, para sumar los números **1 y 2** en notación polaca se escribe como **+ 1 2**, en contraste con la notación infija donde se expresaría como **1 + 2**. En expresiones más complejas, los operadores siguen precediendo a los operandos, pero estos últimos pueden ser expresiones que incluyen, a su vez, operadores y sus propios operandos.

Por ejemplo, la expresión que se escribiría en notación infija convencional como

(5 −6) × 7

se puede escribir en notación polaca como

× 5 - 6 7

EJEMPLO 1

A continuación, usaremos Jupyter, para implementar el código, dando un ejemplo básico de cómo se puede aplicar la notación polaco a algoritmos, utilizando el Lenguaje de JavaScript.

```
class Nodo {
    constructor(value) {
        this.value = value; // Valor del nodo
        this.left = null;   // Hijo izquierdo
        this.right = null;  // Hijo derecho
    }
}

function construirArbol(expresion) {
    const pila = []; // Inicializamos una variable vacía para almacenar los resultados.

    for (const token of expresion) {
        if (!isNaN(token)) { // Si el token es un número
            pila.push(new Nodo(parseInt(token)));
        } else { // Si el token es un operador
            const dcha = pila.pop();
            const izq = pila.pop();
            const nodo = new Nodo(token);
            nodo.left = izq;
            nodo.right = dcha;
            pila.push(nodo);
        }
    }

    return pila.pop(); // Retonar el  último elemento en la pila del árbol construido
}
```

```
function evaluar(nodo) {
    if (!isNaN(nodo.value)) { // Si el valor del nodo es un número
        return nodo.value;
    }

    const izq = evaluar(nodo.left);
    const dcha = evaluar(nodo.right);

    switch (nodo.value) { // Dependiendo del operador evaluar.
        case '+': // Suma
            return izq + dcha;
        case '-': // Resta
            return izq - dcha;
        case '*': // Multiplicación
            return izq * dcha;
        case '/': // División
            return izq / dcha;
        default:
            return null; // Si no hay un operador reconocido, nos dará un valor Nulo
    }
}
```

```
// Expresiones en notación polaca
const expresion1 = ['5', '2', '4', '*', '+', '7', '-'];
const expresion2 = ['6', '2', '*', '3', '5', '*', '+', '18', '-'];
const expresion3 = ['8', '4', '2', '/', '/'];


// Construimos los árboles y evaluar las expresiones
const arbolExpresion1 = construirArbol(expresion1);
const arbolExpresion2 = construirArbol(expresion2);
const arbolExpresion3 = construirArbol(expresion3);
const resultado1 = evaluar(arbolExpresion1);
const resultado2 = evaluar(arbolExpresion2);
const resultado3 = evaluar(arbolExpresion3);

console.log("\nEl resultado de la Expresion 1 es: " + resultado1);
console.log("\nEl resultado de la Expresion 2 es: " + resultado2);
console.log("\nEl resultado de la Expresion 3 es: " + resultado3);
```

Como resultado nos da lo siguiente:

```
El resultado de la Expresion 1 es: 6

El resultado de la Expresion 2 es: 9

El resultado de la Expresion 3 es: 4
```

Explicación detallada del resultado del Programa.

1. **Expresión 1**: **['5', '2', '4', '*', '+', '7', '-']**
   - Se toma el primer elemento '5' y se inserta en la pila.
   - Luego, se toma el siguiente elemento '2' y se inserta en la pila.
   - Después, se toma '4', que es un número, se inserta en la pila.
   - Luego, se encuentra el operador '*', por lo que se toma el último número '4' y el anterior '2', se evalúa la multiplicación (4 * 2 = 8), y se inserta el resultado en la pila.
   - La pila ahora contiene ['5', 8].
   - A continuación, se toma el operador '+', se toma el último número '8' y el anterior '5', se evalúa la suma (5 + 8 = 13), y se inserta el resultado en la pila.
   - La pila ahora contiene [13].
   - Por último, se toma el operador '-', se toma el último número '13' y el anterior '7', se evalúa la resta (13 - 7 = 6), y se inserta el resultado en la pila.
   - Resultado:
   - El resultado final es 6.

2. **Expresión 2**: **['6', '2', '*', '3', '5', '*', '+', '18', '-']**
    - Paso a paso:
    - Se toma el primer elemento '6' y se inserta en la pila.
    - Luego, se toma el siguiente elemento '2' y se inserta en la pila.
    - Después, se encuentra el operador '*', por lo que se toma el último número '2' y el anterior '6', se evalúa la multiplicación (6 * 2 = 12), y se inserta el resultado en la pila.
    - La pila ahora contiene [12].
    - Luego, se toma '3', que es un número, se inserta en la pila.
    - Luego, se toma '5', que es un número, se inserta en la pila.
    - Después, se encuentra el operador '*', por lo que se toma el último número '5' y el anterior '3', se evalúa la multiplicación (3 * 5 = 15), y se inserta el resultado en la pila.
    - La pila ahora contiene [12, 15].
    - A continuación, se toma el operador '+', se toma el último número '15' y el anterior '12', se evalúa la suma (12 + 15 = 27), y se inserta el resultado en la pila.
    - La pila ahora contiene [27].
    - Por último, se toma el operador '-', se toma el último número '27' y el anterior '18', se evalúa la resta (27 - 18 = 9), y se inserta el resultado en la pila.
    - Resultado:
    - El resultado final es 9.
3. **Expresión 3**: **['8', '4', '2', '/', '/']**
    - Paso a paso:
    - Se toma el primer elemento '8' y se inserta en la pila.
    - Luego, se toma el siguiente elemento '4' y se inserta en la pila.
    - Después, se toma '2', que es un número, se inserta en la pila.
    - Luego, se encuentra el operador '/', por lo que se toma el último número '2' y el anterior '4', se evalúa la división (4 / 2 = 2), y se inserta el resultado en la pila.
    - La pila ahora contiene ['8', 2].
    - A continuación, se encuentra el operador '/', por lo que se toma el último número '2' y el anterior '8', se evalúa la división (8 / 2 = 4), y se inserta el resultado en la pila.
    - Resultado:
    - El resultado final es 4.

7. Conclusiones

- La Búsqueda en Anchura y la Búsqueda en Profundidad son dos técnicas esenciales en la exploración y búsqueda de información en estructuras de datos como grafos y árboles.

- La Búsqueda en Anchura para Grafos no Ponderados, es especialmente efectivo en grafos no ponderados, se destaca por encontrar rutas más cortas o soluciones óptimas.

- La Búsqueda en Profundidad para exploración, es eficaz cuando se necesita explorar en profundidad en estructuras de datos complejas, sobre todo en grafos con múltiples niveles de profundidad.

- Al implementar algoritmos de búsqueda, es importante considerar la posibilidad de ciclos, especialmente cuando se va a utilizar la Búsqueda en Profundidad.

- Los algoritmos de búsqueda en árboles y grafos tienen aplicaciones amplias. Desde optimizar sistemas de navegación hasta analizar relaciones en redes sociales y exploraciones en estructuras de datos complejas, su utilidad abarca en diferentes campos de las ciencias de la computación.