Universidad San Carlos de Guatemala
Facultad de ingeniería
escuela de matemáticas

proyecto

Natalia Suceli Garrido Montenegro
202201385

Índice

Revisión bibliográfica **4**
Búsqueda de anchura “BFS” 4
Búsqueda a lo largo o profundidad “DFS” 5
Implementación y experimentación **5**
Búsqueda de anchura “BFS” 5
implementación en python 6
implementación en java 7
Implementación 8
resolución de laberintos 8
Algoritmos de ruta más corta 10
Aplicaciones 11
resolución de problemas con estructuras implícitas 11
enrutamiento de cables 11
Búsqueda en redes sociales 12
Búsqueda de profundidad DFS 12
Implementación en python 12
Implementación de DFS en java 13
Implementación 14
ordenamiento topológico 15
determinación de ciclos 16
aplicación de DFS 17
Sistema de archivos distribuidos 17
Replicación de archivos en servidores 17
Garbage Collection 17
Entrevistas y encuestas **18**
entrevista 18
Comparación **19**
conclusiones **21**
Bibliografía **22**
Anexos **23**

Revisión bibliográfica

Búsqueda de anchura “BFS”

el algoritmo a la búsqueda de anchura se basa en conceptos que son parecidos a algoritmos esenciales como los son el algoritmo de Prim o el algoritmo de Dijkstra Considerando un gráfico G = (V, E) y un vértice con un origen específico “s”, la búsqueda en anchura examina de manera metódica los bordes de G con el objetivo de “*encontra*r” todos los vértices a los que se puede llegar desde s.similar a enviar en todas las direcciones muchos exploradores que recorren todo el gráfico de una manera coordinada.

Este proceso también determina la distancia (el número más pequeño de aristas) desde "s" hasta cada uno de los vértices a los que se puede llegar. Es decir se procede en rondas y se divide los vértices en niveles comenzando en este vértice "s" que vendría siendo el nivel 0 a manera de ir en rondas. en la primera ronda se visita todos los vértices adyacentes al origen "s" En la segunda ronda se debe visitar a 2 aristas del origen siguiendo este proceso hasta tocar todos los vértices

.

Estos resultados dependen del orden que se visita a los vértices adyacentes creando distintas variaciones dependiendo del origen y cómo visitarlo pero las distancia de la búsqueda no varía.

Su tiempo del procedimiento puede variar según el número de vértices y número de aristas en el gráfico. ya que debe pasar por cada vértice al menos una vez siendo bastante eficiente en grafos de gran tamaño

Búsqueda a lo largo o profundidad “DFS”

donde el grafo es conexo. Este también es útil para probar varias propiedades de los gráficos. Comenzamos en un origen o vértice inicial específico. donde se sigue un orden visitando vértices. En caso que un camino nos lleve a un vértice ya visitado nos dirigimos al siguiente no visitado hasta llegar a un “callejón sin salida” es decir que todos los vértices adyacentes ya estén visitados. En este punto se hace un retroceso a lo largo del camino ya encontrado. al volver a los vértices anteriores se hace el mismo cálculo de elegir el el vértice adyacente en el orden dado. Entonces como venimos de un vértice ya visitado tocaría elegir el vértice adyacente más cercano en orden hasta llegar al callejón sin salida y volver a retroceder. hasta estar devuelta en el vértice inicial. y no hay más caminos a explorar. se puede tomar como si se quisiera explorar un laberinto con muchos callejones sin salida donde se tiene que retroceder hasta un punto dado.

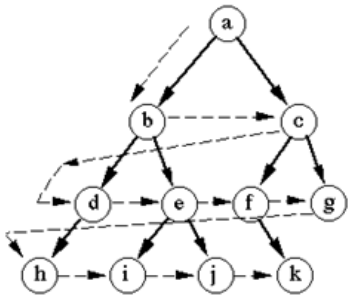
el árbol puede ser que forma puede ser

- aristas de retroceso que conectan los hijos con las hojas
- aristas hacia adelante . que conectan los vértices con sus hijos
- aristas cruzadas que conectan con vértices que no son ni ancestros ni descendiente
- no puede haber 2 vértices del mismo nivel que estén conectados entre sí

Implementación y experimentación

Búsqueda de anchura “BFS”

su uso en gráficas se puede ver algo así



Breadth-first search

implementación en python

```
# Importamos la librería collections para usar la estructura de
datos deque
import collections

# Definimos la función bfs
def bfs(graph, root):
    # Creamos un conjunto para los nodos visitados y una cola para
los nodos por visitar
    visited, queue = set(), collections.deque([root])
    # Agregamos el nodo raíz a los nodos visitados
    visited.add(root)

    # Mientras la cola no esté vacía
    while queue:
        # Sacamos un nodo de la cola
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # Para cada vecino del nodo actual
        for neighbour in graph[vertex]:
            # Si el vecino no ha sido visitado
            if neighbour not in visited:
                # Lo agregamos a los nodos visitados y a la cola
                visited.add(neighbour)
                queue.append(neighbour)

if __name__ == '__main__':
    # Código principal
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("El recorrido en amplitud es: ")
    bfs(graph, 0)
```

Este código implementa el algoritmo de búsqueda en amplitud (BFS) en Python. El algoritmo BFS se utiliza para recorrer grafos o árboles. Recorrer significa visitar cada nodo del grafo. BFS es un algoritmo recursivo para buscar todos los vértices

de un grafo o un árbol. La complejidad temporal del algoritmo BFS se representa en forma de O(V + E), donde V es el número de nodos y E es el número de aristas.

implementación en java

```
// Importamos las clases necesarias
import java.io.*;
import java.util.*;

// Definimos la clase Graph
class Graph {
    private int V;   // Número de vértices
    private LinkedList<Integer> adj[]; // Lista de adyacencia

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Función para agregar una arista al grafo
    void addEdge(int v,int w) {
        adj[v].add(w);
    }

    // Implementación del método BFS
    void BFS(int s) {
        // Marca todos los vértices como no visitados (por defecto)
        boolean visited[] = new boolean[V];

        // Crea una cola para BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Marca el nodo actual como visitado y lo encola
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0) {
            // Desencola un vértice de la cola y lo imprime
            s = queue.poll();
            System.out.print(s+" ");

            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
```

```
    // Método principal
    public static void main(String args[]) {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("El recorrido en amplitud es: ");

        g.BFS(2);
    }
}
```

Este código implementa el algoritmo de búsqueda en amplitud (BFS) en Java. El algoritmo BFS se utiliza para recorrer grafos o árboles. Recorrer significa visitar cada nodo del grafo. BFS es un algoritmo recursivo para buscar todos los vértices de un grafo o un árbol

Implementación

resolución de laberintos

la idea es encontrar un camino desde un inicio hasta un final. donde la búsqueda inicia en el inicio del laberinto. donde se explora el laberinto de manera uniforme en todas las direcciones donde se encontrara el camino más corto. donde este puede resolver cualquier laberinto que se representa como un gráfico. donde se toma como una cuadrícula los caminos y paredes donde cada celda es un nodo si el algoritmo puede llegar al nodo de salida entonces existe un camino a través del laberinto siendo estos caminos de los más cortos posibles.

Al implementar un algoritmo en un código como lo es java el método de resolución seria el siguiente

```
import java.util.*;

class Cell {
    int x, y;
```

```
    Cell(int x,int y){
        this.x = x;
        this.y = y;
    }
}

public class MazeSolver {
    static int[][] dirs={{0, 1}, {0, -1}, {-1, 0}, {1, 0}};

    public static void solve(int[][] maze, Cell start, Cell end) {
        int m = maze.length;
        int n = maze[0].length;
        boolean[][] visited = new boolean[m][n];
        Queue<Cell> queue = new LinkedList<>();
        queue.offer(start);
        visited[start.x][start.y] = true;

        while (!queue.isEmpty()) {
            Cell cell = queue.poll();
            if (cell.x == end.x && cell.y == end.y) {
                System.out.println("Solución encontrada en: " +
cell.x + " " + cell.y);
                return;
            }
            for (int[] dir : dirs) {
                int newX = cell.x + dir[0];
                int newY = cell.y + dir[1];
                if (newX >= 0 && newY >= 0 && newX < m && newY < n
&& !visited[newX][newY] && maze[newX][newY] == 1) {
                    queue.offer(new Cell(newX, newY));
                    visited[newX][newY] = true;
                }
            }
        }
        System.out.println("No se encontró solución");
    }

    public static void main(String[] args) {
        int[][] maze = {
            {1, 1, 1, 1},
            {1, 0, 1, 0},
            {0, 1, 1, 1},
            {1, 1, 0, 1}
        };
        Cell start = new Cell(0, 0);
        Cell end = new Cell(3, 3);
        solve(maze, start, end);
    }
}
```

Este código define una clase Cell para representar las celdas del laberinto. Luego define una función solve que utiliza BFS para encontrar el camino más corto desde la celda de inicio hasta la celda final. Si encuentra una solución, imprime las coordenadas de la celda final. Si no encuentra una solución, imprime un mensaje indicando que no se encontró ninguna solución

Algoritmos de ruta más corta

Comienza en la raíz del árbol (o algún nodo arbitrario de un grafo, a veces denominado “clave de búsqueda”) y explora primero los nodos vecinos antes de pasar a los vecinos del siguiente nivel.

- Utiliza una cola para realizar la búsqueda en anchura. Marca el vértice de origen como descubierto y lo agrega a la cola
- Mientras la cola no esté vacía, quita el nodo frontal de la cola. Para cada vecino del nodo actual, si el vecino no ha sido descubierto, lo marca como descubierto y lo agrega a la cola

Este algoritmo garantiza que encuentres el camino más corto en términos del número de aristas en los caminos.Ejemplo de implementación en C++

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

// Estructura de datos para almacenar una arista de la  gráfica
struct Edge {
    int src, dest;
};

// Una clase para representar un objeto graph
class Graph {
public:
    // un vector de vectores para representar una lista de
adyacencia al vertice
    vector<vector<int>> adjList;

    // Constructor de graph
    Graph(vector<Edge> const &edges, int n) {
        // cambiar el tamaño del vector para contener `n` elementos
de tipo `vector<int>`
        adjList.resize(n);

        // agrega aristas al grafo no dirigido
        for (auto &edge: edges) {
            adjList[edge.src].push_back(edge.dest);
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Realiza BFS en el grafo a partir del vértice `v`
void BFS(Graph const &graph, int v, vector<bool> &discovered) {
    // crea una queue para hacer BFS
    queue<int> q;

    // marca el vértice de origen como descubierto
    discovered[v] = true;
```

```
    // poner en queue el vértice fuente
    q.push(v);

    // bucle hasta que la queue esté vacía
    while (!q.empty()) {
        // quitar la queue del nodo frontal e imprimirlo
        v = q.front();
        q.pop();
        cout << v << " ";

        // hace lo mismo para todos los vértices adyacentes al
vértice actual `v`
        for (int u : graph.adjList[v]) {
            if (!discovered[u]) {
                // marca como descubierto y poner en queue
                discovered[u] = true;
                q.push(u);
            }
        }
    }
}
```

.

Aplicaciones

resolución de problemas con estructuras implícitas

Donde se encuentra un espacio de estado. y se puede usar el BFS explorando estos espacios de manera sistemática. como ya dijiste esta los laberintos otro uso es en juegos como el ajedrez donde se puede intuir un árbol del juego a partir de una posición encontrando los caminos posibles desde la posición siendo uno de estos el ganador

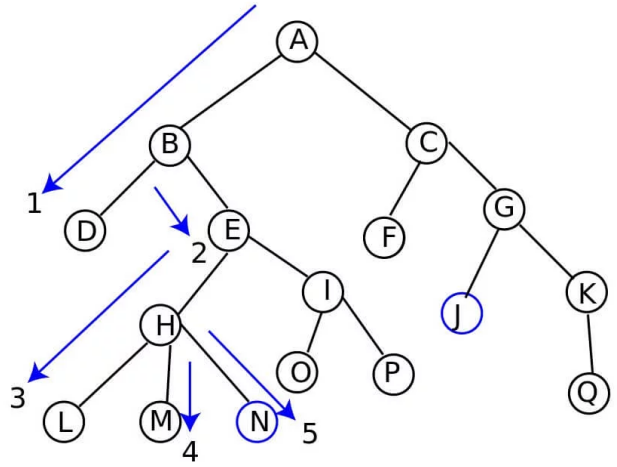
enrutamiento de cables

Desarrollado por C.Y Lee en 1961 utilizado en la industria electrónica para trazar las rutas óptimas para los cables de un circuito integrado. para minimizar la longitud del cableado y evitar intersecciones entre ellos. donde se modela el problema como un grafo donde los nodos representan las ubicaciones posibles de los cables y las aristas representan las posibles conexiones entre ellos. Luego con BFS encuentra el camino más corto entre los nodos.

Búsqueda en redes sociales

Las conexiones de las redes sociales pueden modelarse como grafos donde los nodos representan a los usuarios y las aristas las relaciones entre ellos. con el BFS donde se puede buscar amigos para compartir entre otros. encontrando amigos por amigos .

Búsqueda de profundidad DFS



Aquí se puede ver como se mira un árbol DFS donde hay un orden alfabético ascendente donde se elige primero el más corto y se hace presente los retrocesos al elegir primero de por ser el más corto a b luego se retrocede y se elige e al encontrar un camino sin salida.

Implementación en python

```
# Definimos la función DFS
def dfs(grafo, nodo_inicio):
    # Creamos un conjunto para almacenar los nodos visitados
    visitados = set()

    # Creamos una pila para almacenar los nodos por visitar
    pila = [nodo_inicio]

    while pila:
        # Tomamos el último nodo de la pila
        nodo = pila.pop()

        # Si el nodo no ha sido visitado
        if nodo not in visitados:
            # Lo marcamos como visitado
            visitados.add(nodo)

            # Obtenemos los vecinos del nodo
            vecinos = grafo[nodo]
```

```
            # Agregamos los vecinos no visitados a la pila
            pila.extend(vecino for vecino in vecinos if vecino not
in visitados)

    return visitados

# Definimos un grafo de prueba
grafo = {
  'A': ['B', 'C'],
  'B': ['A', 'D', 'E'],
  'C': ['A', 'F'],
  'D': ['B'],
  'E': ['B', 'F'],
  'F': ['C', 'E']
}

# Llamamos a la función DFS con el grafo de prueba y un nodo de
inicio
print(dfs(grafo, 'A'))  # Imprime: {'A', 'B', 'C', 'D', 'E', 'F'}
```

Esta implementación utiliza una estructura de datos de pila para almacenar los nodos que aún no se han explorado. Comienza con el nodo de inicio y explora tan profundamente como sea posible a lo largo de cada rama antes de retroceder. El algoritmo continúa hasta que se han explorado todos los nodos accesibles desde el nodo de inicio.

Implementación de DFS en java

```
import java.util.*;

public class DFS {
    // Creamos una clase para representar el grafo
    static class Grafo {
        int numVertices;
        LinkedList<Integer>[] adjList;

        // Constructor del grafo
        Grafo(int numVertices) {
            this.numVertices = numVertices;
            adjList = new LinkedList[numVertices];

            for (int i = 0; i < numVertices; i++) {
                adjList[i] = new LinkedList<>();
            }
        }

        // Método para agregar una arista al grafo
        void agregarArista(int src, int dest) {
            adjList[src].add(dest);
        }
```

```
        // Método para realizar la búsqueda DFS
        void DFS(int v, boolean[] visitado) {
            // Marcamos el nodo actual como visitado
            visitado[v] = true;
            System.out.print(v + " ");

            // Visitamos todos los vértices adyacentes a este nodo
            Iterator<Integer> i = adjList[v].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visitado[n])
                    DFS(n, visitado);
            }
        }

        // Método para iniciar la búsqueda DFS
        void iniciarDFS(int v) {
            // Creamos un arreglo para almacenar los nodos
visitados
            boolean[] visitado = new boolean[numVertices];

            // Llamamos al método DFS
            DFS(v, visitado);
        }
    }

    public static void main(String args[]) {
        Grafo g = new Grafo(4);

        g.agregarArista(0, 1);
        g.agregarArista(0, 2);
        g.agregarArista(1, 2);
        g.agregarArista(2, 0);
        g.agregarArista(2, 3);
        g.agregarArista(3, 3);

        System.out.println("Búsqueda en Profundidad (DFS) desde el
vértice 2:");

        g.iniciarDFS(2);
    }
}
```

Esta implementación utiliza una estructura de datos de pila implícita a través de la recursión para almacenar los nodos que aún no se han explorado. Comienza con el nodo de inicio y explora tan profundamente como sea posible a lo largo de cada rama antes de retroceder.

Implementación

ordenamiento topológico

Se usa el algoritmo para ordenar topológicamente usado en la programación de tareas

```
from collections import defaultdict

# Clase para representar un grafo
class Grafo:
    def __init__(self, vertices):
        # Número de vértices
        self.grafo = defaultdict(list)
        self.V = vertices

    # Añadir una arista al grafo
    def agregarArista(self, u, v):
        self.grafo[u].append(v)

    # Función recursiva para el ordenamiento topológico
    def topologicalSortUtil(self, v, visitado, pila):

        # Marcar el nodo actual como visitado
        visitado[v] = True

        # Recorrer todos los vértices adyacentes a este vértice
        for i in self.grafo[v]:
            if visitado[i] == False:
                self.topologicalSortUtil(i, visitado, pila)

        # Agregar el vértice actual a la pila que almacena el
resultado
        pila.insert(0, v)

    # Función para el ordenamiento topológico
    def topologicalSort(self):

        # Marcar todos los vértices como no visitados
        visitado = [False]*self.V
        pila = []

        # Llamar a la función recursiva para todos los vértices no
visitados
        for i in range(self.V):
            if visitado[i] == False:
                self.topologicalSortUtil(i, visitado, pila)

        # Imprimir el ordenamiento topológico
        print(pila)

# Crear un grafo de prueba
g = Grafo(6)
g.agregarArista(5, 2)
g.agregarArista(5, 0)
g.agregarArista(4, 0)
g.agregarArista(4, 1)
g.agregarArista(2, 3)
g.agregarArista(3, 1)

print("Ordenamiento Topológico del grafo:")
g.topologicalSort()
```

Este código crea un grafo y luego llama a la función topologicalSort para obtener un ordenamiento topológico de sus vértices. La función topologicalSort utiliza una pila para almacenar el resultado y un arreglo para marcar los vértices visitados. Para cada vértice no visitado, llama a la función topologicalSortUtil, que explora recursivamente todos los vértices adyacentes y luego agrega el vértice actual a la pila.

determinación de ciclos

```
from collections import defaultdict

class Grafo:
    def __init__(self, vertices):
        self.V = vertices
        self.grafo = defaultdict(list)

    def agregarArista(self, u, v):
        self.grafo[u].append(v)
        self.grafo[v].append(u)

    def tieneCicloAux(self, v, visitado, padre):
        # Marcar el nodo actual como visitado
        visitado[v] = True

        # Recorrer todos los nodos vecinos
        for i in self.grafo[v]:
            # Si el nodo vecino no ha sido visitado, entonces
recursivamente verificar si tiene ciclo
            if visitado[i] == False:
                if self.tieneCicloAux(i, visitado, v):
                    return True
            # Si un nodo vecino ya ha sido visitado y no es el
padre del nodo actual, entonces hay un ciclo
            elif padre != i:
                return True

        return False

    def tieneCiclo(self):
        # Marcar todos los nodos como no visitados
        visitado = [False] * (self.V)

        # Llamar a la función auxiliar recursiva para detectar
ciclos en diferentes componentes conectados
        for i in range(self.V):
            if visitado[i] == False:
                if self.tieneCicloAux(i, visitado, -1) == True:
                    return True

        return False

# Crear un grafo de prueba
g1 = Grafo(5)
```

```
g1.agregarArista(1, 0)
g1.agregarArista(1, 2)
g1.agregarArista(2, 0)
g1.agregarArista(0, 3)
g1.agregarArista(3, 4)

if g1.tieneCiclo():
    print("El grafo contiene un ciclo")
else:
    print("El grafo no contiene un ciclo")
```

aplicación de DFS

Sistema de archivos distribuidos

se utiliza la DFS en los sistemas de archivos distribuidos que usa una tecnología de almacenamiento de información atribuida de manera de Big Data

Replicación de archivos en servidores

Se usa para la replicación de archivos en servidores como lo usa windows server que permite replicar eficientemente carpetas a través de múltiples servidores y sitios.

Garbage Collection

Cuando se tiene un código en lenguajes como Java, Python entre otros se usa para detectar objetos que no están en uso para poder ser eliminados y liberar memoria

árboles en notación polaca

Entrevistas y encuestas

entrevista

1. ¿Qué estructura de datos utiliza BFS?

   BFS utiliza una estructura de datos de cola.

2. ¿Qué algoritmo es más eficiente cuando la solución está muy lejos del vértice de origen?

   DFS es más eficiente cuando la solución se encuentra en algún lugar profundo de un árbol o lejos del vértice de origen.

3. ¿Cuál de los siguientes algoritmos se basa en bordes?

   DFS es un algoritmo basado en bordes.

4. ¿Cuál de los siguientes algoritmos es más adecuado para árboles de decisión utilizados en juegos o rompecabezas?

   DFS es más adecuado para problemas de juegos o rompecabezas.

5. ¿Cuál de los siguientes algoritmos trabaja con el concepto de FIFO (Primero en Entrar, Primero en Salir)?

   BSF trabaja con el concepto de FIFO (Primero en Entrar, Primero en Salir)

Comparación

primero otros aspectos a comparar que hay entre ellos seria

	BFS	DFS
base	se centra en visitar cada vértice del grado antes de pasar al siguiente nivel	se centra en visitar cada arista del grafo profundizando tanto como sea posible antes de retroceder
estructura de datos	usa datos de cola junto con el concepto FIFO	usa datos de recursión así que usa el concepto LIFO
espacio de memoria	no es tan efectiva ya que necesita mantener el registro de todos los nodos del árbol a la vez	es más eficiente ya que no necesita mantener un registro de todos los nodos al árbol a la vez
enfoque utilizado	trabaja con el concepto FIFO esto significa que visita primero todos los vecinos antes de pasar al siguiente nivel	usa el concepto LIFO esto significa que visita primero un nodo luego explora tanto como sea posible cada rama antes de retroceder
adecuación árboles de decisión	primero considera a todos los vecinos por lo cual no es adecuado para árboles de decisión utilizados en juegos o rompecabezas	es más adecuado a juegos de rompecabezas se toma la decisión y luego se explora todo los caminos a través de esa decisión
eliminación de nodos recorridos	los nodos se recorren varias veces y se eliminan de la cola	los nodos visitados se añaden a una pila y luego se eliminan cuando no hay más nodos para visitar

	BFS	DFS
eficiencia	Es eficiente cuando la solución no está muy lejos del vértice de origen. si el árbol es amplio usa demasiada memoria	es eficiente cuando la solución se encuentra en algún lugar profundo de el árbol o lejos del origen y
complejidad	su complejidad de tiempo es menor ya que almacena los nodos y abarca más rápido	tiende a ser más larga porque si la solución está al inicio se debe hacer un largo retroceso
aplicabilidad	se usa para encontrar el camino mas corto entre dos nodos . también se utiliza para encontrar el grafo bipartido o calcular el flujo máximo de la red	se usa para encontrar los componentes conectados al grafo o realizar ordenamientos topológico, así como encontrar puentes en un grafo

conclusiones

1. Los algoritmos de BFS y DFS son dos métodos fundamentales para explorar árboles y grafos. donde BFS explora todos los nodos a una determinada profundidad antes de pasar al siguiente nivel. mientras que DFS explora profundamente a lo largo de cada rama antes de retroceder. Donde ambos algoritmos tienen sus propias ventajas y desventajas donde su uso dependerá del problema que se esté resolviendo
2. tienen una amplio campo de aplicaciones donde en las ares de computación. donde BFS se usa para encontrar el camino más corto y el DFS para ver los componentes conectados o ordenar de manera topológica. Estas aplicaciones se pueden llegar a extender en otras áreas como lo es la administración de big data o redes.

3. Cada método de búsqueda puede llegar a tener mas eficiencia dependiendo del tamaño de búsqueda o en su caso la profundidad de este grafo.

4. según expertos estos métodos son muy son bastantes útiles pero de la misma manera tienen bastantes limitaciones. ejemplo BFS consume bastante memoria si el árbol es muy amplio. con DFS puede llevar mucho tiempo si el árbol es muy profundo y la solución estaba en un retroceso de los primeros niveles.

5. Aunque se puede tomar como su teoría muy simple es de notar su utilidad en varios aspectos de la computación ya que puede existir muchas variantes de algoritmos según el lenguaje o el uso que deseemos darle. ya que son más útiles para problemas específicos de búsqueda o orden establecido.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press

- Drozdek, A. (2004). Data Structures and Algorithms in Java (2nd ed.). Cengage Learning

- Techie Delight. (2023). Breadth-First Search (BFS) - Techie Delight. Recuperado el 9 de octubre de 2023, de https://www.techiedelight.com/breadth-first-search/

- Khan Academy. (s.f.). Búsqueda en anchura y sus usos. Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/breadth-first-search-and-its-uses

Anexos

In [*]:
```
def calcular(s):
    pila = []
    operadores = ['+', '-', '*', '/']
    for c in s:
        if c not in operadores:
            pila.append(int(c))
        else:
            top1 = pila.pop()
            top2 = pila.pop()
            if c == '+':
                pila.append(top2 + top1)
            elif c == '-':
                pila.append(top2 - top1)
            elif c == '*':
                pila.append(top2 * top1)
            elif c == '/':
                pila.append(int(top2 / top1))
    return pila.pop()

while True:
    s = input("Introduce una expresión en notación polaca inversa: ")
    print(calcular(s.split()))
```

```
Introduce una expresión en notación polaca inversa: 5 2 * 8 + 10 *
180
Introduce una expresión en notación polaca inversa: 5 3 * 15 + 20 +
50
Introduce una expresión en notación polaca inversa: 8 8 * 6 +
70

Introduce una expresión en notación polaca inversa:
```