UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS
DEPARTAMENTO DE MATEMÁTICA

MATEMÁTICA PARA COMPUTACIÓN 2
Ing. José Alfredo González Díaz
Aux. Byron Estuardo
Sección A

Nombre	Tema	
Isamir Alessandro Armas Cano	Algoritmos de Arbol	
Registro Académico	**Actividad**	**Fecha**
201901403	Proyecto	30 / 10 / 2023

DESCRIPCIÓN DE CALIFICACIÓN	
Presentación y formato (10)	
Procedimiento paso a paso (50)	
Ejercicios Calificados (40)	
CALIFICACIÓN TOTAL (100)	

INDICE

Algoritmos de búsqueda en árboles por ancho y por profundidad

Introducción

En la actualidad, las nuevas tecnologías están produciendo un cambio importante en nuestra sociedad. La educación no puede mantenerse al margen de este desarrollo y los avances tecnológicos deberían generar un cambio sustancial en nuestros sistemas de enseñanza. Por otra parte, la cantidad de estímulos que el alumno está recibiendo de su entorno hacen que, por contraste, la clase y el estudio tradicionales resulten menos motivadores. En este contexto, se impone la necesidad de introducir nuevos alicientes en el proceso de enseñanza–aprendizaje y las nuevas tecnologías proporcionan el instrumento idóneo para generar el tipo de estímulo que nuestros estudiantes necesitan para involucrarse de lleno en dicho proceso. Los algoritmos de grafos, estudiados en el programa de Matemática Discreta de la titulación de Ingeniería Informática, son, probablemente, los primeros algoritmos de cierta complejidad a los que el alumno va a enfrentarse en el curso de sus estudios. Esto los hace particularmente importantes, ya que la manera en que el estudiante los afronte va a influir, positiva o negativamente, en su formación posterior. Los algoritmos de grafos son eminentemente visuales, por lo que, para su comprensión y aprendizaje, es de gran ayuda un entorno gráfico que complemente las explicaciones teóricas. Aun así, los visualizadores gráficos, independientemente de lo bien que estén presentados, tienen poco valor si no consiguen involucrar al estudiante en su propio proceso de aprendizaje. Por esta razón, es esencial que el alumno no permanezca como mero espectador del desarrollo del algoritmo, sino que participe activamente en él. Este es el objetivo principal que inspiró el diseño de nuestras herramientas en general, y de la presentada en este trabajo en particular, ya que es el usuario quien debe decidir, en cada paso, el siguiente nodo que va a ser visitado. Gracias a su flexibilidad –admiten que el usuario introduzca su propio grafo–, a su interactividad –es el usuario quien va ejecutando los algoritmos mientras el programa comprueba su desarrollo– y a su presentación gráfica –la diferenciación de colores permite saber en qué paso del algoritmo nos encontramos en cada momento–, los tutoriales disponibles en nuestro sitio web, y en particular el dedicado a los algoritmos de búsqueda, resultan muy adecuados, tanto para complementar la exposición del profesor, como para facilitar el aprendizaje del alumno mediante la práctica de los algoritmos.

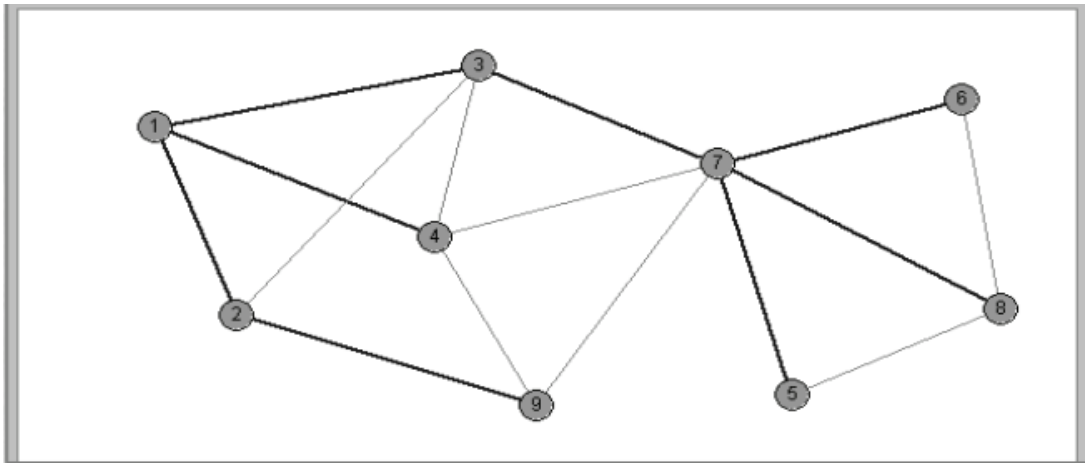
Los algoritmos de búsqueda en anchura y en profundidad

Estos algoritmos tienen su origen en el estudio de los laberintos. El primero en aparecer es el algoritmo de búsqueda en profundidad: Lucas (1882) [4] menciona que Tremaux, en su algoritmo, ya aplicaba esta idea y Tarry (1895) 574 Recursos docentes [16] también la utilizaba en la resolución de laberintos, aunque la versión actual aparece mucho más tarde, en la publicación de Tarjan (1972) [15]. El algoritmo de búsqueda en anchura es utilizado por Moore (1959) [6] en la obtención del camino más corto en un laberinto. La importancia de estos dos algoritmos radica en sus múltiples aplicaciones: búsqueda de un vértice de características específicas, estudio de conexión o cálculo del número de componentes conexas, obtención del camino más corto en grafos no ponderados, búsqueda de ciclos, búsqueda de puntos de articulación y comprobación de la propiedad de biconexión en grafos, ordenación topológica en grafos dirigidos acíclicos, obtención de caminos hamiltonianos, búsqueda en bases de datos o redes, etc. Otra cualidad, así mismo importante, es el hecho de que ofrecen un ejemplo, dentro de unos algoritmos relativamente sencillos, de dos estrategias de amplia utilización en informática: FIFO

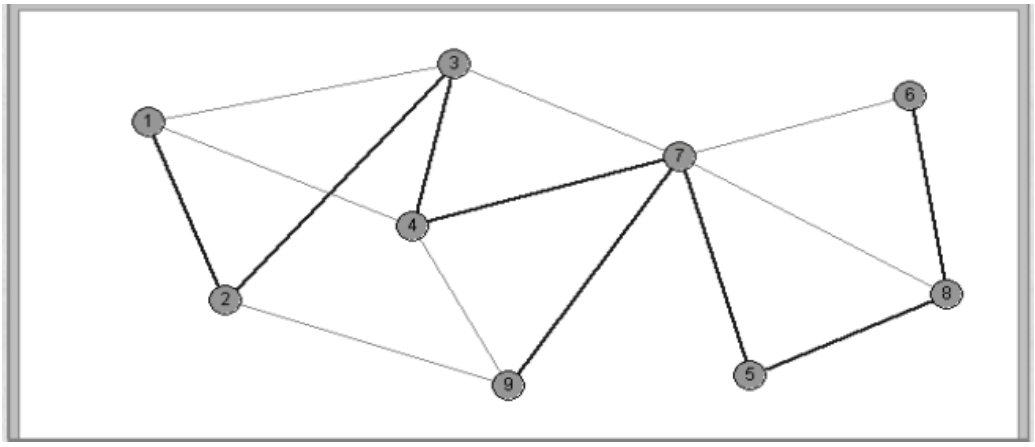
(First In First Out) o estructura de almacenamiento en cola y LIFO (Last In First Out) o estructura de almacenamiento en pila. Debido a esta última característica, es muy importante, desde el punto de vista pedagógico, que el alumno de primer curso comprenda en profundidad estos algoritmos y se familiarice con el funcionamiento de estas dos estrategias, ya que tendrá que hacer repetido uso de ellas durante su formación y, muy probablemente, en su futuro profesional. También es importante que el alumno sea consciente de cómo esta diferencia de estrategia, que aparece como una pequeña modificación en el algoritmo, influye en el desarrollo de éste y se habitúe a estudiar a fondo todos los algoritmos con los que se encuentre, ya que un pequeño detalle puede tener consecuencias substanciales en el resultado final.

Los algoritmos de búsqueda en anchura y profundidad

A los tutoriales para la enseñanza de grafos se accede desde el sitio web del D.M.A.1 y los enlaces están situados en la página de applets de java relacionados con matemática discreta2 La página de entrada a los algoritmos de búsquedas3 contiene una pequeña introducción al tema de grafos (lo necesario para la comprensión de los algoritmos), la descripción de dichos algoritmos, las instrucciones de uso de la herramienta y un enlace a la página que contiene el applet propiamente dicho4 . El programa está diseñado para permitir al usuario dibujar sus propios grafos, dándole la posibilidad de deshacer tantas veces como sea necesario. Una vez introducido, es posible ejecutar ambos algoritmos sobre el mismo grafo, consiguiendo, de esta forma, destacar las diferencias entre los árboles resultantes al aplicar cada uno de ellos.



Búsqueda por anchura



Búsqueda por profundidad

Tipos de algoritmos de búsqueda de arboles

Búsqueda en Árbol Binario

- Búsqueda en Árbol Binario de Búsqueda (BST): Este es un tipo de árbol binario en el que se garantiza que los nodos izquierdos son menores que el nodo raíz y los nodos derechos son mayores. El algoritmo de búsqueda en BST es eficiente y suele utilizarse para buscar elementos en estructuras ordenadas.

- Recorrido en Orden (Inorder Traversal): Este algoritmo visita los nodos en un BST en orden ascendente. Es útil para obtener una lista de elementos en orden.

- Recorrido en Preorden (Preorder Traversal): Visita primero el nodo raíz, luego los nodos izquierdos y, finalmente, los nodos derechos.

- Recorrido en Postorden (Postorder Traversal): Visita primero los nodos izquierdos y derechos, y luego el nodo raíz.

Árboles N-arios

- Búsqueda en Árbol N-ario: Los árboles n-arios son estructuras en las que cada nodo puede tener más de dos hijos. Los algoritmos de búsqueda en estos árboles suelen basarse en la recursión y pueden adaptarse a diferentes aplicaciones.

Árboles de Decisión

- Árboles de Decisión: Se utilizan en aprendizaje automático para la clasificación y toma de decisiones. Los algoritmos, como el algoritmo ID3, C4.5 y CART, construyen árboles de decisión basados en características de datos de entrenamiento.

4. Árboles AVL:

- Árboles AVL: Estos son árboles binarios de búsqueda equilibrados que garantizan que la altura de los subárboles izquierdo y derecho en cada nodo difiera en un valor máximo de uno. Los algoritmos de búsqueda en árboles AVL son eficientes y garantizan un tiempo de búsqueda equilibrado.

5. Árboles B:

- Árboles B: Se utilizan en bases de datos y sistemas de archivos para mejorar la eficiencia en la búsqueda y la inserción. Los árboles B son árboles equilibrados que tienen múltiples hijos por nodo y son ideales para almacenar grandes cantidades de datos.

6. Árboles Trie:

- Árboles Trie: También conocidos como "árboles de prefijo", se utilizan comúnmente en estructuras de datos para representar diccionarios y buscar palabras con prefijos comunes. Estos árboles son especialmente útiles para la búsqueda de palabras en motores de búsqueda y sistemas de recomendación.

7. Árboles de Segmento:

- Árboles de Segmento: Estos árboles se utilizan para realizar consultas en rangos en secuencias, como encontrar el mínimo o la suma en un rango específico de una matriz. Son útiles en problemas de búsqueda de intervalos en algoritmos.

Cada uno de estos algoritmos de búsqueda de árboles tiene sus propias ventajas y desventajas y se adapta mejor a diferentes tipos de problemas. La elección del algoritmo dependerá de la naturaleza de los datos y la aplicación específica. La comprensión de estos algoritmos es esencial para cualquier persona que trabaje en ciencias de la computación o en campos relacionados.

Implementación y experimentación

comparación básica de los algoritmos de búsqueda en árboles en Python, utilizando una estructura de árbol simple y las implementaciones de búsqueda en anchura (BFS) y búsqueda en profundidad (DFS). En este ejemplo, utilizaremos una clase Nodo para representar los nodos del árbol y luego implementaremos ambas estrategias de búsqueda.

```
from collections import deque
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.hijos = []

    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)
# Construcción del árbol de ejemplo
raiz = Nodo(1)
raiz.agregar_hijo(Nodo(2))
raiz.agregar_hijo(Nodo(3))
raiz.hijos[0].agregar_hijo(Nodo(4))
raiz.hijos[0].agregar_hijo(Nodo(5))
raiz.hijos[1].agregar_hijo(Nodo(6))
```

Implementamos la búsqueda en anchura:

```
def bfs(raiz):
    resultado = []
    cola = deque([raiz])

    while cola:
```

```
        nodo = cola.popleft()
        resultado.append(nodo.valor)
        cola.extend(nodo.hijos)

    return resultado
```
Implementamos la búsqueda en profundidad:
```
def dfs(raiz):
    resultado = []

    def dfs_recursivo(nodo):
        if nodo:
            resultado.append(nodo.valor)
            for hijo in nodo.hijos:
                dfs_recursivo(hijo)

    dfs_recursivo(raiz)
    return resultado
print("Búsqueda en Anchura (BFS):", bfs(raiz))
print("Búsqueda en Profundidad (DFS):", dfs(raiz))
```

El resultado del programa es el siguiente:

Búsqueda en Anchura (BFS): [1, 2, 3, 4, 5, 6]
Búsqueda en Profundidad (DFS): [1, 2, 4, 5, 3, 6]

Análisis y comparación

Claro, proporcionaré un análisis detallado del programa que implementa la búsqueda en anchura (BFS) y la búsqueda en profundidad (DFS) en un árbol utilizando Python.

Primero, la implementación incluye una clase `Nodo` que se utiliza para representar los nodos del árbol. Esta clase tiene dos atributos:

- `valor`: Almacena el valor del nodo.

- `hijos`: Almacena una lista de los nodos hijos de este nodo.

Luego, se crea un árbol de ejemplo. En este caso, el árbol tiene la siguiente estructura:

```
    1
   / \
  2   3
 / \
4   5
  /
 6
```

El nodo raíz tiene un valor de 1, dos hijos (2 y 3), y el nodo 2 tiene dos hijos (4 y 5), mientras que el nodo 5 tiene un hijo (6).

La búsqueda en anchura (BFS) se implementa utilizando una cola (deque) para rastrear los nodos a medida que se exploran. Comienza desde la raíz y, por cada nodo visitado, se agregan sus hijos a la cola en orden. Luego, se repite el proceso hasta que la cola esté vacía.

La búsqueda en profundidad (DFS) se implementa de manera recursiva. Comienza desde la raíz y visita un nodo, luego explora recursivamente todos sus hijos antes de pasar al siguiente hijo del nodo actual. Esto se repite hasta que todos los nodos se visitan.

Finalmente, se invocan ambas funciones para realizar las búsquedas en el árbol de ejemplo, y los resultados se imprimen en la salida estándar. La salida mostrará los valores de los nodos en el orden en que se encontraron utilizando cada estrategia de búsqueda.

Este programa proporciona una base sólida para comprender cómo funcionan las estrategias de búsqueda en árboles (BFS y DFS) y cómo se pueden implementar en Python. Puedes experimentar con diferentes estructuras de árbol y tamaños para ver cómo cambian los resultados de las búsquedas.

Comparación

Búsqueda en Anchura (BFS - Breadth-First Search):

BFS es un algoritmo de búsqueda en árboles que comienza en el nodo raíz y explora todos los nodos vecinos a la misma profundidad antes de avanzar a los nodos a la siguiente profundidad. El BFS se puede implementar de manera iterativa utilizando una cola.

Búsqueda en Profundidad (DFS - Depth-First Search):

DFS es un algoritmo de búsqueda en árboles que explora un camino hasta el final antes de retroceder y explorar otros caminos. Puede implementarse de manera recursiva o utilizando una pila (stack).

Diferencias clave entre BFS y DFS:

-Orden de exploración: BFS explora en orden de anchura, mientras que DFS explora en profundidad.

-Estructura de datos utilizada: BFS utiliza una cola, y DFS puede usar recursión o una pila.

-Espacio requerido: En general, BFS tiende a utilizar más memoria que DFS debido a la cola.

La elección entre BFS y DFS depende de la naturaleza del problema y de lo que se busca en el árbol. Por ejemplo, si estás buscando la ruta más corta entre dos nodos en un grafo no ponderado, BFS es una elección adecuada. Si estás explorando todas las soluciones posibles en un espacio de búsqueda, DFS puede ser más apropiado.

Aplicaciones a las ciencias de computación

Los algoritmos de búsqueda en árboles por anchura (BFS) y por profundidad (DFS) son fundamentales en la ciencia de la computación y tienen numerosas aplicaciones en una variedad de campos.

Búsqueda en Anchura (BFS):

Búsqueda de caminos más cortos: BFS se utiliza para encontrar el camino más corto entre dos nodos en un grafo no ponderado. Esto es útil en la navegación de mapas, enrutamiento de redes y algoritmos de búsqueda en laberintos.

Grafos bipartitos: BFS se utiliza para determinar si un grafo es bipartito, es decir, si sus nodos pueden dividirse en dos conjuntos de tal manera que no haya aristas que conecten nodos del mismo conjunto. Esto es útil en problemas de emparejamiento y en la teoría de grafos.

Búsqueda en la web: En motores de búsqueda y rastreo web, BFS se utiliza para descubrir y visitar páginas web en un patrón de búsqueda estructurado.

Algoritmos de flujo máximo: BFS se utiliza en algoritmos como el de Edmonds-Karp para encontrar el flujo máximo en una red de flujo. El flujo máximo tiene aplicaciones en la asignación óptima de recursos en redes de transporte.

Resolución de laberintos y juegos: BFS se utiliza en la resolución de laberintos y en juegos como el Taquin y el Cubo de Rubik, para encontrar la solución más corta.

Búsqueda en Profundidad (DFS):

Recorridos en árboles y grafos: DFS se utiliza para recorrer árboles y grafos en profundidad. Es esencial para la exploración exhaustiva de estructuras de datos jerárquicas, como árboles de decisión, árboles AVL y grafos.

Backtracking y fuerza bruta: DFS es fundamental en algoritmos de backtracking para encontrar todas las soluciones posibles a un problema, como el problema de las N reinas, la generación de permutaciones y la resolución de rompecabezas.

Búsqueda en profundidad limitada: En inteligencia artificial, se utiliza DFS con una profundidad máxima para explorar árboles de búsqueda de estados en juegos como el ajedrez o el Go.

Topología y componentes fuertemente conexos: DFS se utiliza en la determinación de componentes fuertemente conexos en grafos dirigidos, lo que es útil en el análisis de redes y grafos.

Ordenamiento topológico: DFS se utiliza para realizar un ordenamiento topológico en un grafo dirigido acíclico (DAG). Esto es esencial en tareas como la planificación de proyectos y la compilación de código.

Arboles de notación polaca

Árboles de Notación Polaca (Polish Notation Trees)

Los árboles de notación polaca son estructuras de datos utilizadas para representar expresiones matemáticas en notación polaca inversa (NPI) o notación polaca, también conocida como RPN (Reverse Polish Notation). Esta notación es un enfoque de escritura de expresiones matemáticas en el que los operadores se colocan después de sus operandos en lugar de entre ellos, como es común en la notación infija. Los árboles de notación polaca son una forma de representar y evaluar expresiones escritas en notación polaca inversa.

Elementos clave en la notación polaca inversa (RPN):

- Operandos: Números en la expresión.

- Operadores: Símbolos que indican una operación (por ejemplo, +, -, *, /).

- No se requieren paréntesis en la notación polaca inversa, ya que la notación misma indica el orden de las operaciones.

Estructura de los árboles de notación polaca:

- Cada nodo en el árbol es un operador o un operando.

- Los nodos hoja son operandos.

- Los nodos internos son operadores.

- Los árboles de notación polaca son árboles binarios de expresiones, donde cada nodo interno tiene exactamente dos hijos (izquierdo y derecho).

Construcción de árboles de notación polaca:

- Dada una expresión en notación polaca inversa, se puede construir un árbol de notación polaca recorriendo la expresión de izquierda a derecha.

- Cuando se encuentra un operando, se crea un nodo hoja y se coloca en la pila.

- Cuando se encuentra un operador, se crean dos nodos hijos a partir de los operandos en la parte superior de la pila y se apilan como un nuevo nodo interno.

Ejemplo de árbol de notación polaca:

Supongamos que tenemos la expresión en notación polaca inversa: "3 4 + 2 *". Podemos construir un árbol de notación polaca de la siguiente manera:

```
    *
   / \
  +   2
 / \
3   4
```

Este árbol representa la expresión (3 + 4) * 2 en notación infija.

Aplicaciones de árboles de notación polaca:

- Evaluación eficiente de expresiones matemáticas.

- Cálculos en calculadoras y lenguajes de programación de notación polaca inversa (como Forth y PostScript).

- Optimización de compiladores y evaluación de expresiones en lenguajes de programación.

- Implementación de estructuras de datos para procesar expresiones matemáticas.

En resumen, los árboles de notación polaca son una estructura de datos importante para representar y evaluar expresiones matemáticas en notación polaca inversa. Permiten una evaluación eficiente de expresiones y tienen aplicaciones en calculadoras, compiladores y lenguajes de programación.

Ejemplo utilizando Jupyter

Crear una función para evaluar expresiones en notación polaca:

Aquí hay una función de ejemplo que evalúa expresiones en notación polaca. Esta función recibe una cadena de texto que representa la expresión y devuelve el resultado.

```
def evaluar_notacion_polaca(expresion):
    pila = []
    operadores = {'+', '-', '*', '/'}
    elementos = expresion.split()

    for elemento in elementos:
        if elemento not in operadores:
            pila.append(elemento)
        else:
            operando2 = pila.pop()
            operando1 = pila.pop()
            resultado = eval(f'{operando1} {elemento} {operando2}')
            pila.append(str(resultado))

    return float(pila[0])
```

Evaluar expresiones de notación polaca:

```
expresion1 = "3 4 +"
expresion2 = "5 2 * 8 +"
expresion3 = "7 3 / 2 *"

resultado1 = evaluar_notacion_polaca(expresion1)
resultado2 = evaluar_notacion_polaca(expresion2)
resultado3 = evaluar_notacion_polaca(expresion3)

resultado1, resultado2, resultado3
```

Mostrar los resultados:

(7.0, 18.0, 4.666666666666667)

Conclusiones

1. **BFS es ideal para encontrar rutas más cortas:** El algoritmo de búsqueda en anchura (BFS) es muy eficiente para encontrar la ruta más corta entre dos nodos en un grafo no ponderado. Esto lo hace valioso en aplicaciones de navegación, como GPS y la planificación de rutas.

2. **DFS es útil en la exploración exhaustiva:** La búsqueda en profundidad (DFS) es especialmente útil cuando se necesita explorar exhaustivamente un espacio de búsqueda o un grafo. Puede ayudar a encontrar todas las soluciones a un problema o recorrer un árbol de decisión en aplicaciones de inteligencia artificial.

3. **BFS requiere más memoria:** En comparación con DFS, BFS tiende a requerir más memoria, ya que debe mantener una cola de nodos por explorar. Esto puede ser una consideración crítica en aplicaciones con restricciones de memoria.

4. **DFS es más fácil de implementar de manera recursiva:** DFS se presta bien a la implementación recursiva, lo que puede hacer que el código sea más sencillo y elegante. Por otro lado, BFS se implementa de manera iterativa utilizando una cola.

5. **Elección depende del problema y la estructura de datos:** La elección entre BFS y DFS depende de la naturaleza del problema y la estructura de datos en la que estés trabajando. Cada algoritmo tiene sus propias ventajas y desventajas, y la elección adecuada puede marcar la diferencia en la eficiencia y la eficacia de la solución.

6. **BFS es útil en problemas de grafos ponderados:** Si estás trabajando con grafos ponderados y deseas encontrar la ruta más corta entre dos nodos, BFS puede adaptarse utilizando una cola de prioridad (en lugar de una cola estándar) para garantizar que los nodos se visiten en orden de peso creciente. Esto se conoce como "Búsqueda en Anchura con Cola de Prioridad" y se usa en el algoritmo de Dijkstra.

7. **DFS es valioso en aplicaciones de retroceso:** En problemas de optimización y búsqueda, DFS se usa comúnmente en algoritmos de retroceso, donde se exploran todas las posibles soluciones de manera exhaustiva. Es crucial en áreas como la resolución de rompecabezas, planificación de horarios y optimización de rutas.

8**. Combinación de técnicas:** En muchos casos, la elección no es exclusiva entre BFS y DFS. Puedes combinar ambas técnicas en un solo algoritmo, dependiendo de las necesidades. Por ejemplo, el algoritmo A* utiliza BFS con una función de costo heurística para encontrar rutas eficientes en grafos ponderados.

Bibliografía

-Sánchez Torrubia, M. G., & Gutiérrez Revenga, S. (2006). Tutorial interactivo para la enseñanza y el aprendizaje de los algoritmos de búsqueda en anchura y en profundidad.

-"Introduction to Algorithms" (Introducción a los algoritmos) de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein: Este libro, a menudo llamado "CLRS" por las iniciales de los autores, es un recurso ampliamente utilizado en cursos de algoritmos y cubre una variedad de algoritmos, incluyendo algoritmos de búsqueda en árboles.

- Millán, M. E. (1994). Comparación de métodos de búsqueda.

- Arias-Méndez, E., & Torres-Rojas, F. (2017). Comparación de rutas metabólicas mediante algoritmos de evaluación simples. In Memorias de congresos TEC.