			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:	**Nelson Andres Santa Cruz Gil**	Carné:	**202202728**

Proyecto#1

Algoritmos de búsqueda en arboles por ancho y profundidad:

Ancho: La Búsqueda en Anchura (BFS) es un algoritmo de búsqueda que se utiliza en ciencias de la computación y matemáticas para explorar o recorrer gráficos y árboles. Su enfoque principal es la exploración sistemática en capas, lo que significa que comienza desde un nodo raíz y se expande gradualmente a los nodos vecinos en niveles sucesivos. El orden de exploración se asemeja a una onda que se propaga a través del grafo, visitando primero todos los nodos en el mismo nivel antes de avanzar al siguiente nivel.

BFS se implementa utilizando una estructura de datos llamada "cola" (queue) para llevar un seguimiento de los nodos que aún no se han visitado. El algoritmo se inicia en el nodo raíz y coloca este nodo en la cola. Luego, comienza a extraer nodos de la cola uno por uno, los visita y agrega sus nodos vecinos no visitados a la cola. Este proceso continúa hasta que se han visitado todos los nodos o se ha encontrado el nodo de destino (si la búsqueda tiene un objetivo).

Uno de los usos más comunes de BFS es encontrar el camino más corto entre dos nodos en un grafo no ponderado, lo que lo hace ideal para problemas como la búsqueda en mapas o la navegación de sitios web. También se utiliza en la búsqueda sistemática de soluciones en

problemas que involucran la exploración de estados, como en juegos y problemas de optimización.

La ventaja clave de BFS es que garantiza encontrar la solución óptima en términos de la menor cantidad de pasos o movimientos. Sin embargo, en grafo densos, puede ocupar mucha memoria, ya que requiere almacenar todos los nodos en el nivel actual en la cola antes de avanzar al siguiente nivel.

Profundidad: La Búsqueda en Profundidad (DFS) es otro algoritmo de búsqueda utilizado en la exploración de gráficos y árboles, pero a diferencia de BFS, se enfoca en explorar en profundidad antes de retroceder. Comienza en un nodo raíz y se adentra en el grafo visitando recursivamente los nodos hijos hasta que alcanza un nodo terminal. Luego, regresa al nodo predecesor y continúa explorando otros caminos.

DFS no utiliza una estructura de datos adicional para mantener un registro de los nodos no visitados, en lugar de eso, utiliza una pila de llamadas para mantener un seguimiento de los nodos a visitar. Cuando un nodo es visitado, se coloca en la pila y se exploran sus nodos hijos. Si no hay más nodos que visitar en una rama específica, DFS regresa al nodo predecesor y explora otros caminos. Este proceso continúa hasta que se ha explorado todo el grafo o se ha encontrado el nodo de destino (si la búsqueda tiene un objetivo).

DFS se utiliza en situaciones en las que se necesita explorar exhaustivamente todas las posibles soluciones o verificar la conectividad entre nodos en un grafo. Por ejemplo, se utiliza para encontrar todos los caminos posibles en un laberinto o en el análisis de redes de computadoras.

Aplicaciones a las ciencias de la computación

BFS (Búsqueda en Anchura):

Encontrar el camino más corto en redes: BFS se utiliza en algoritmos de enrutamiento de redes para encontrar la ruta más corta entre dos nodos, medida por el número de nodos intermedios. Esto es esencial en aplicaciones de redes de computadoras y navegación GPS.

Grafos bipartitos: BFS se utiliza para probar si un grafo es bipartito, es decir, si sus nodos pueden dividirse en dos conjuntos disjuntos de tal manera que no haya aristas que conecten nodos dentro del mismo conjunto. Esto es útil en problemas de asignación y diseño de algoritmos.

Árbol de expansión mínimo: En grafos no ponderados, BFS se utiliza para encontrar un árbol de expansión mínimo, que es un subgrafo que conecta todos los nodos con el menor número de aristas posible. Esto tiene aplicaciones en problemas de diseño de redes y optimización.

Rastreadores web: Los rastreadores web utilizan BFS para recorrer sitios web y seguir enlaces de manera sistemática. Esto es fundamental en motores de búsqueda y en la recopilación de información en línea.

DFS (Búsqueda en Profundidad):

Resolución de rompecabezas y juegos: DFS se utiliza para explorar todas las posibles soluciones en juegos y rompecabezas, como el ajedrez, el sudoku y los laberintos. También es esencial en inteligencia artificial para la toma de decisiones en juegos.

Recorrido de árboles y grafos: En la programación y la manipulación de estructuras de datos como árboles binarios, DFS se utiliza para realizar recorridos en profundidad en árboles, como el preorden, inorden y postorden. Esto es fundamental en la manipulación de estructuras de datos y análisis de algoritmos.

Búsqueda de componentes conectados: DFS se utiliza para encontrar componentes conectados en un grafo, lo que es útil en la identificación de grupos de nodos interconectados en redes sociales, análisis de redes y detección de comunidades.

Ordenamiento topológico: DFS se aplica en la resolución de problemas que implican un grafo dirigido acíclico (DAG) para realizar un ordenamiento topológico de los nodos. Esto es crucial en tareas como la planificación de proyectos y la programación de tareas.

Recolección de basura: En la gestión de la memoria de programas informáticos, DFS se utiliza en algoritmos de recolección de basura para identificar y liberar memoria no utilizada.

Serialización/Deserialización de estructuras de datos: DFS se utiliza en la serialización y deserialización de estructuras de datos, lo que permite convertir estructuras de datos en un formato que se puede almacenar o transmitir y luego reconstruirlos.

Algoritmo para arboles de notación polaca:

Las notaciones de prefijo (o polaca, en homenaje a Jan Łukasiewicz), de infijo y de postfijo (o polaca inversa) son formas de escritura de expresiones algebraicas que se diferencian por la posición relativa que toman los operadores y los operandos. En la notación de prefijo, el operador se escribe delante de los operandos (+ 3 4), entre los operandos en la notación de infijo (3 + 4) y tras los operandos en la de posfijo (3 4 +). La notación de prefijo fue propuesta en 1924 por el matemático, lógico y filósofo polaco Jan Łukasiewicz (1878-1956), de allí el nombre alternativo por la que se conoce. Macho Stadler, M. (2019).

Tomando en cuenta lo anterior mencionado se desarrollo el siguiente algoritmo que tiene la funcionalidad de:

1. Dividir la expresión en tokens (números y operadores) que se colocan en una estructura de datos, generalmente una pila.
2. Recorrer la lista de procesos uno por uno.
3. Cuando se encuentra un número, se coloca en la pila.
4. Cuando se encuentra un operador, se extraen uno o más números de la pila y se aplica la operación correspondiente. El resultado se coloca de nuevo en la pila.
5. El proceso se repite hasta que se ha recorrido toda la expresión y al final del proceso, el valor restante en la pila es el resultado de la expresión.

Solución en Jupyter:

In [2]:
```
def evaluar_notacion_polaca(expresion):
    pila = []
    operadores = set(['+', '-', '*', '/'])

    for token in expresion.split():
        if token not in operadores:
            pila.append(float(token))
        else:
            b = pila.pop()
            a = pila.pop()
            if token == '+':
                resultado = a + b
            elif token == '-':
                resultado = a - b
            elif token == '*':
                resultado = a * b
            elif token == '/':
                if b == 0:
                    raise ValueError("División por cero")
                resultado = a / b
            pila.append(resultado)

    if len(pila) == 1:
        return pila[0]
    else:
        raise ValueError("Expresión no válida")

expresion1 = "3 4 + 2 *"
expresion2 = "5 1 2 + 4 * + 3 -"
expresion3 = "2 3 * 5 4 * + 9 -"

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

print(f"Resultado 1: {resultado1}")
print(f"Resultado 2: {resultado2}")
print(f"Resultado 3: {resultado3}")
```

```
Resultado 1: 14.0
Resultado 2: 14.0
Resultado 3: 17.0
```

In [4]:
```python
def evaluar_notacion_polaca2(expresion):
    pila = []
    operadores = set(['+', '-', '*', '/'])

    for token in expresion.split():
        if token not in operadores:
            pila.append(float(token))
        else:
            b = pila.pop()
            a = pila.pop()
            if token == '+':
                resultado = a + b
            elif token == '-':
                resultado = a - b
            elif token == '*':
                resultado = a * b
            elif token == '/':
                if b == 0:
                    raise ValueError("División por cero")
                resultado = a / b
            pila.append(resultado)

    if len(pila) == 1:
        return pila[0]
    else:
        raise ValueError("Expresión no válida")

expresion1 = "4 5 + 2 3 + *"
expresion2 = "6 7 + 2 3 + * 2 /"
expresion3 = "8 2 / 4 6 - * 3 +"

resultado1 = evaluar_notacion_polaca2(expresion1)
resultado2 = evaluar_notacion_polaca2(expresion2)
resultado3 = evaluar_notacion_polaca2(expresion3)

print(f"Resultado 1: {resultado1}")
print(f"Resultado 2: {resultado2}")
print(f"Resultado 3: {resultado3}")
```

```
Resultado 1: 45.0
Resultado 2: 32.5
Resultado 3: -5.0
```

In [7]:
```python
def evaluar_notacion_polaca3(expresion):
    pila = []
    operadores = set(['+', '-', '*', '/'])

    for token in expresion.split():
        if token not in operadores:
            pila.append(float(token))
        else:
            b = pila.pop()
            a = pila.pop()
            if token == '+':
                resultado = a + b
            elif token == '-':
                resultado = a - b
            elif token == '*':
                resultado = a * b
            elif token == '/':
                if b == 0:
                    raise ValueError("División por cero")
                resultado = a / b
            pila.append(resultado)

    if len(pila) == 1:
        return pila[0]
    else:
        raise ValueError("Expresión no válida")


expresion1 = "3 4 + 5 6 - *"
expresion2 = "2 7 / 3 8 + *"
expresion3 = "9 2 * 4 5 - /"

resultado1 = evaluar_notacion_polaca3(expresion1)
resultado2 = evaluar_notacion_polaca3(expresion2)
resultado3 = evaluar_notacion_polaca3(expresion3)

print(f"Resultado 1: {resultado1}")
print(f"Resultado 2: {resultado2}")
print(f"Resultado 3: {resultado3}")
```

```
Resultado 1: -7.0
Resultado 2: 3.142857142857143
Resultado 3: -18.0
```

Conclusiones:

- Tanto la búsqueda en profundidad (DFS) como la búsqueda en anchura (BFS) son algoritmos esenciales en la ciencia de la computación para abordar una amplia variedad de problemas. Estos algoritmos proporcionan enfoques sistemáticos para la exploración y resolución de problemas en ámbitos que van desde la optimización de rutas en aplicaciones de navegación hasta la gestión de memoria en programas informáticos.

- Estos procedimientos de búsqueda son los motores fundamentales de búsqueda web que hoy en día se utilizan para rastrear y organizar información en línea, así como la resolución de juegos y rompecabezas, lo cual nos da una nueva perspectiva respecto a su importancia en la sociedad.

- La elección adecuada entre DFS y BFS depende de la naturaleza de un problema. Mientras que BFS es ideal para encontrar el camino más corto en redes y sistemas, DFS es fundamental para explorar exhaustivamente todas las soluciones posibles en problemas múltiples, por lo cual es fundamental saber sus diferencias para su aplicación.

- Se puede decir que los DFS y BFS juegan un papel crucial en el desarrollo de nuevas tecnologías como la inteligencia artificial, ya que por ejemplo se emplea DFS en algoritmos de búsqueda informada, y BFS es fundamentalmente en algoritmos de búsqueda no informada.

- Gracias al descubrimiento de estos procedimientos en la actualidad es posible la organización eficiente de actividades complejas hasta proyectos simples, debido a esto en el presente tienen un impacto directo en la calidad de vida de las personas día con día.

Bibliografía:

- Cespedes Alcocer Cristian, Lipa Challapa Jorge, Ramirez Iriarte Elvis R., Lopez Choque Franz A. (2013). Búsqueda por Amplitud.Recuperado de: https://slidedeck.io/jorgelipa/pst-busqueda-amplitud

- López Mamani, M. (2020). DFS vs BFS. Recuperado de: https://www.encora.com/es/blog/dfs-vs-bfs

- Difference between BFS and DFS. (2023). GeeksforGeeks..Recuperado de: https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

- Codecademy Team. (2021). Tree Traversal: Breadth-First Search vs Depth-First Search. Codecademy. https://www.codecademy.com/article/tree-traversal

- Macho Stadler, M. (2019). La notación polaca, la de Jan Łukasiewicz. Cuaderno de Cultura Científica. Recuperado de https://culturacientifica.com/2019/02/13/la-notacion-polaca-la-de-jan-lukasiewicz/

- Tong, X., & Tang, Z. (2023). BFS & DFS. University of Illinois Urbana-Champaign. https://courses.engr.illinois.edu/cs225/fa2023/resources/bfs-dfs/

- Coto, E. (2003). Algoritmos Básicos de Grafos. Universidad Central de Venezuela, Facultad de Ciencias, Escuela de Computación, Laboratorio de Computación Gráfica (LCG). Venezuela, Caracas, Apdo. 47002, 1041-A.