Universidad de San Carlos de Guatemala

Facultad de ingeniería

Aux. Edgar Daniel Cil

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

Nombre: Renato Yazid Vicente Gómez

Carnet: 202202462

CUI: 3637484580102

Asesorado por el ingeniero José Alfredo González Díaz

Algoritmos de búsqueda en arboles por ancho y profundidad

Los algoritmos de búsqueda en árboles por anchura y por profundidad son técnicas fundamentales en la ciencia de la computación y se utilizan para recorrer y buscar información en estructuras de datos como los árboles y los grafos. Ambos algoritmos son fundamentales en la búsqueda y exploración de estructuras de datos arbóreas y se utilizan según las necesidades específicas de un problema.

Algoritmo de búsqueda en arboles por ancho:

Se trata de un algoritmo de búsqueda que se inicia en el nodo raíz de un grafo y a partir de ahí explora todos los nodos vecinos. Posteriormente, se expande a los vecinos de esos nodos, y este proceso se repite hasta que se haya recorrido todo el grafo. Es importante tomar en cuenta que, si se encuentra el nodo de interés antes de explorar todo el grafo, el proceso de búsqueda concluye.

La búsqueda por anchura es utilizada en situaciones en las que es crucial seleccionar la mejor opción en cada etapa del recorrido.

Una búsqueda por medio del algoritmo de búsqueda a lo ancho se encarga de explorar los nodos de un grafo desde la raíz, eligiendo un nodo inicial y extendiéndose a través de sus vecinos. Este método se caracteriza por explorar todos los vecinos de un nodo antes de avanzar a los siguientes niveles en el grafo. Inicia su recorrido desde la raíz y se extiende hacia los nodos cercanos en una secuencia nivelada. Si se encuentra el nodo objetivo antes de recorrer todo el grafo, la búsqueda termina.

Algoritmo de búsqueda en arboles por profundidad:

Una búsqueda en profundidad es un algoritmo de exploración de nodos en un grafo, donde se expande gradualmente cada nodo que se encuentra, siguiendo una ruta recursiva desde el nodo padre hacia sus nodos hijos. Cuando no hay más nodos por visitar en esa ruta, el algoritmo retrocede al nodo anterior y repite el proceso con los nodos vecinos. Es relevante destacar que, si se encuentra el nodo de interés antes de revisar todos los nodos, la búsqueda se da por concluida.

La búsqueda en profundidad se utiliza cuando se necesita evaluar si una de varias soluciones posibles cumple con ciertos requisitos, como en el caso del problema del recorrido del caballo en un tablero de ajedrez, donde se busca encontrar una ruta que pase por las 64 casillas del tablero.

Una búsqueda por medio del algoritmo de búsqueda por profundidad se puede denominar como él un algoritmo simple de utilizar, este algoritmo se utiliza cuando

una lista no está en ningún orden o en orden. Es un método muy simple que puede llegar a ser muy útil cuando la cantidad de datos es muy pequeña y si los datos buscados no están en orden puede ser el único método que puede utilizarse para hacer las respectivas búsquedas.

Aplicaciones a las ciencias de computación

Algoritmo de búsqueda en arboles por ancho:

- Encontrar la ruta más breve entre dos nodos, calculando la cantidad de nodos intermedios.
- Determinar si un conjunto de nodos en un grafo puede dividirse en dos subconjuntos distintos.
- Hallar el árbol de expansión de menor costo en un grafo que no tiene pesos en sus aristas.
- Realizar un rastreo web para indexar páginas y obtener información de la web.
- Desarrollar sistemas de navegación GPS para identificar ubicaciones cercanas y rutas óptimas.

Algoritmo de búsqueda en arboles por profundidad:

- Hallar los nodos interconectados en una estructura de datos gráfica.
- Determinar el orden en que se deben visitar los nodos en un grafo dirigido sin ciclos.
- Identificar conexiones entre puentes en un conjunto de nodos interconectados.
- Resolver problemas que tienen solamente una solución, como laberintos o acertijos.
- Descubrir grupos de nodos que están fuertemente relacionados entre sí en un grafo.

Comparación entre algoritmos de búsqueda en arboles por ancho y profundidad

La selección entre el algoritmo de búsqueda en arboles por ancho y profundidad depende de las características particulares del problema y la forma del árbol o grafo. Cuando se requiere encontrar la solución más próxima al nodo inicial y obtener la ruta más corta en cuanto a la cantidad de conexiones, es mejor utilizar un algoritmo de búsqueda por ancho. Por otro parte, el algoritmo de búsqueda por profundidad resulta beneficioso en situaciones que involucran árboles profundos y cuando el objetivo es hallar una solución sin enfocarse necesariamente en la longitud de la ruta.

Ventajas del algoritmo de búsqueda en arboles por ancho:

El algoritmo de búsqueda por anchura asegura la localización de la solución óptima en un problema de costo menor, lo que significa que una vez que se encuentre una solución, es la mejor posible. También puede implicar expandir todos los nodos en el mismo nivel en el que se encontró la solución para identificar todas las soluciones potenciales.

Encuentra la solución más cercana al nodo raíz, garantizando encontrar la solución más corta en términos de la cantidad de arcos. Puede utilizarse para encontrar todas las soluciones dentro de un árbol o grafo.

Ventajas del algoritmo de búsqueda en arboles por profundidad:

sus requisitos de memoria son modestos, lo que la hace eficiente en términos de uso de recursos. Su implementación es sencilla y directa, lo que la convierte en una opción accesible para una variedad de aplicaciones.

La búsqueda en profundidad tiene una ventaja clave en términos de complejidad espacial en comparación con la búsqueda por anchura, ya que solo mantiene un camino de nodos en la memoria en un momento dado. Esto es especialmente útil en árboles o grafos profundos, donde la búsqueda por anchura podría requerir un espacio significativamente mayor.

La búsqueda en profundidad es completa en ausencia de ciclos repetitivos. Esto significa que siempre encontrará una solución si existe, lo que la convierte en una herramienta confiable para la resolución de problemas en una variedad de contextos.

Desventajas del algoritmo de búsqueda en arboles por ancho:

Uno de los problemas más destacados es su alto orden de complejidad computacional. Esto significa que a medida que los parámetros y la complejidad del problema aumentan, los requerimientos computacionales crecen rápidamente, a menudo llegando a niveles inaceptables**.**

La demanda significativa de memoria que requiere la búsqueda por anchura. Para explorar todos los nodos en el mismo nivel antes de pasar al siguiente, es necesario mantener una gran cantidad de información en la memoria. Esto puede ser un desafío en sistemas con recursos de almacenamiento limitados.

La búsqueda por anchura tiende a tener una complejidad temporal aún mayor, lo que significa que el tiempo necesario para encontrar una solución puede aumentar considerablemente a medida que se profundiza en el árbol de búsqueda.

Desventajas del algoritmo de búsqueda en arboles por profundidad:

Uno de los problemas más notables es su propensión a caer en ciclos infinitos, lo que significa que, en ciertas situaciones, podría continuar explorando el mismo camino sin llegar a una solución. Esta característica puede hacer que la búsqueda en profundidad sea ineficiente en la resolución de problemas específicos.

Otro aspecto a tomar en cuenta es que la búsqueda en profundidad puede encontrar soluciones que están más alejadas de la raíz del árbol de búsqueda que otras alternativas. Esto puede llevar a soluciones no tan optimas, ya que no necesariamente encuentra la ruta más corta o eficiente.

En algunos casos, puede encontrar una solución rápida, pero no necesariamente la mejor solución posible en términos de costo o longitud.

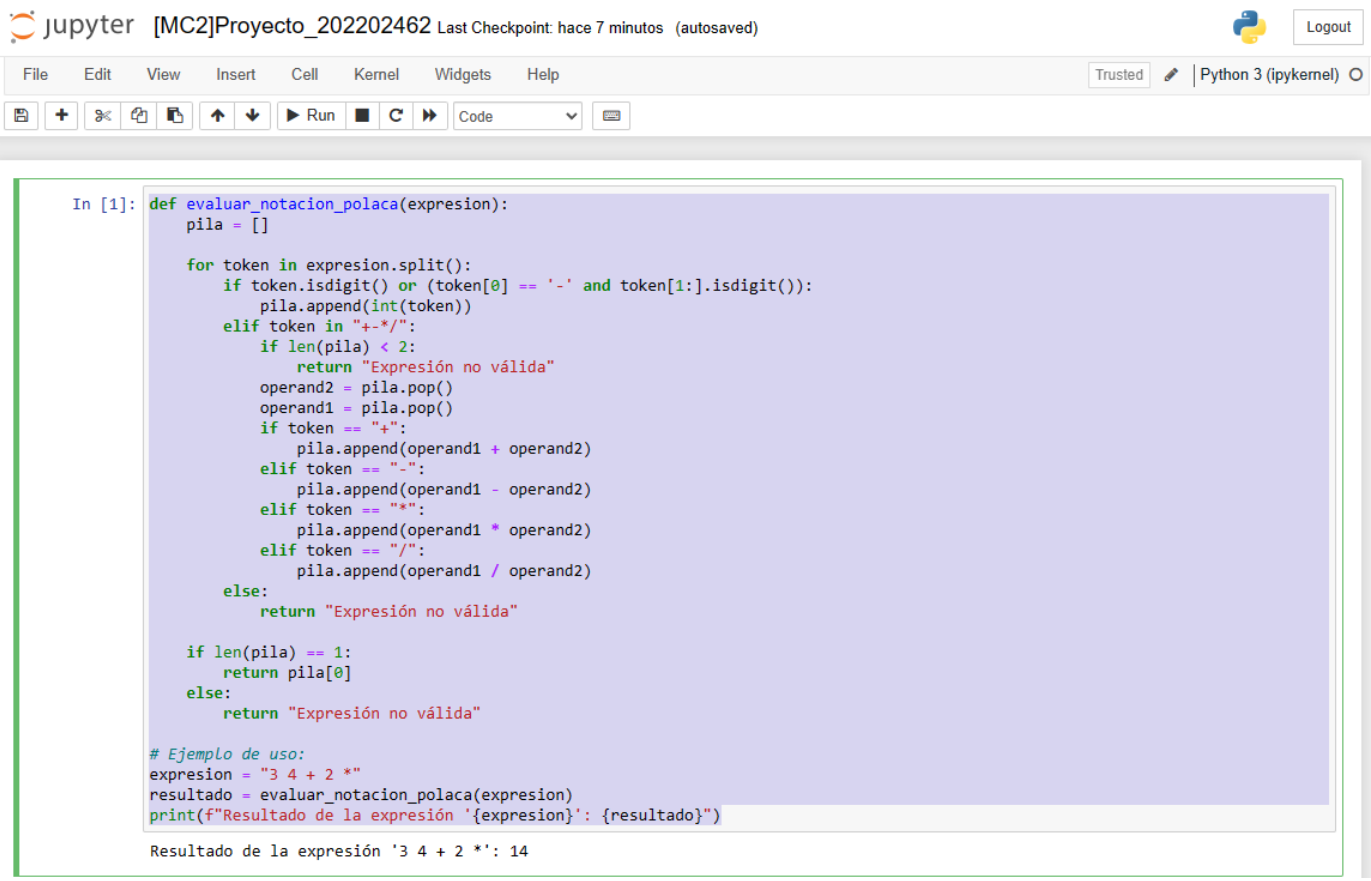
algoritmo para arboles de notación polaca

```
def evaluar_notacion_polaca(expresion):
    pila = []

    for token in expresion.split():
        if token.isdigit() or (token[0] == '-' and token[1:].isdigit()):
            pila.append(int(token))
        elif token in "+-*/":
            if len(pila) < 2:
                return "Expresión no válida"
            operand2 = pila.pop()
            operand1 = pila.pop()
            if token == "+":
                pila.append(operand1 + operand2)
            elif token == "-":
                pila.append(operand1 - operand2)
            elif token == "*":
                pila.append(operand1 * operand2)
            elif token == "/":
                pila.append(operand1 / operand2)
        else:
            return "Expresión no válida"

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

```
# Ejemplo de uso:
expresion = "3 4 + 2 *"
resultado = evaluar_notacion_polaca(expresion)
print(f"Resultado de la expresión '{expresion}': {resultado}")
```



In [9]:
```
def evaluar_notacion_polaca(expresion):
    pila = []

    for token in expresion.split():
        if token.isdigit() or (token[0] == '-' and token[1:].isdigit()):
            pila.append(int(token))
        elif token in "+-*/":
            if len(pila) < 2:
                return "Expresión no válida"
            operand2 = pila.pop()
            operand1 = pila.pop()
            if token == "+":
                pila.append(operand1 + operand2)
            elif token == "-":
                pila.append(operand1 - operand2)
            elif token == "*":
                pila.append(operand1 * operand2)
            elif token == "/":
                pila.append(operand1 / operand2)
        else:
            return "Expresión no válida"

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

# Ejemplo de uso:
expresion = "1 2 + 3 *"
resultado = evaluar_notacion_polaca(expresion)
print(f"Resultado de la expresión '{expresion}': {resultado}")
```

```
Resultado de la expresión '1 2 + 3 *': 9
```

In [ ]:

jupyter [MC2]Proyecto_202202462 Last Checkpoint: hace 12 minutos (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3 (ipykernel)
Run Code

```
Resultado de la expresión '1 2 + 3 *': 9
```

In [10]:
```
def evaluar_notacion_polaca(expresion):
    pila = []

    for token in expresion.split():
        if token.isdigit() or (token[0] == '-' and token[1:].isdigit()):
            pila.append(int(token))
        elif token in "+-*/":
            if len(pila) < 2:
                return "Expresión no válida"
            operand2 = pila.pop()
            operand1 = pila.pop()
            if token == "+":
                pila.append(operand1 + operand2)
            elif token == "-":
                pila.append(operand1 - operand2)
            elif token == "*":
                pila.append(operand1 * operand2)
            elif token == "/":
                pila.append(operand1 / operand2)
        else:
            return "Expresión no válida"

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

# Ejemplo de uso:
expresion = "2 7 - 1 /"
resultado = evaluar_notacion_polaca(expresion)
print(f"Resultado de la expresión '{expresion}': {resultado}")
```

```
Resultado de la expresión '2 7 - 1 /': -5.0
```

In [ ]:

Este algoritmo evalúa expresiones en notación polaca en Jupyter Notebook. Simplemente se debe ingresar la expresión en notación polaca en la variable ‘expresion’, luego esta llama a la función ‘evaluar_notacion_polaca’ y muestra el resultado.

Se crea una lista llamada ‘pila’ que se utilizará para realizar los cálculos y seguir un enfoque de pila para evaluar la expresión.

Se itera a través de la expresión en notación polaca dividiendo la cadena en tokens utilizando ‘.split().’ Cada token representa un número, un operador, o cualquier otro carácter no válido.

Una vez que se ha recorrido toda la expresión y se han realizado los cálculos necesarios, se verifica el estado de la pila y si solo queda un valor en la pila, este es el resultado final de la expresión.

Conclusiones

Cada uno de estos algoritmos son útiles dependiendo del contexto del problema, el algoritmo por anchura se utiliza en problemas que requieren encontrar la solución mas corta y completa, pero necesita de mas espacio o memoria. El algoritmo por profundidad es más eficiente en términos de memoria y más útil en exploraciones mas profundas, pero no siempre tiene la solución mas optima y a veces no es tan completo.

La eficiencia del algoritmo por anchura se observa en su capacidad para encontrar rápidamente una solución, ya que se desplaza de manera ordenada a través del grafo. Además, garantiza la completitud, lo que significa que siempre encontrará una solución si existe una. Esto lo hace especialmente valioso en aplicaciones que requieren respuestas precisas y en tiempo real, como los sistemas de navegación GPS.

El algoritmo de búsqueda por profundidad es capaz de explorar profundamente las diferentes ramas de un grafo, lo que puede ser necesario para encontrar todas las soluciones posibles o para verificar todas las combinaciones en busca de una respuesta deseada. No garantiza encontrar la solución más eficiente o corta, ya que podría quedar atrapado en una rama larga antes de explorar otras alternativas.

El algoritmo de búsqueda por profundidad es valioso en problemas donde se requiere una exploración completa y exhaustiva de todas las posibilidades, a pesar de que la solución encontrada podría no ser la más eficiente. Es una herramienta esencial en campos como la inteligencia artificial.

la eficiencia del algoritmo de búsqueda por anchura puede verse limitada en grafos muy grandes o complejos, ya que requiere mantener una estructura de datos cola que puede consumir una cantidad significativa de memoria. El algoritmo de búsqueda por anchura es valioso en aplicaciones que priorizan la velocidad y la eficiencia, siempre y cuando el tamaño del grafo sea manejable en términos de memoria y capacidad de procesamiento.

Bibliografía

https://slidedeck.io/jorgelipa/pst-busqueda-amplitud

https://www.studocu.com/es-ar/document/universidad-nacional-de-la-rioja/inteligencia-artificial/cuadro-comparativo-busquedas/8438513

https://askanydifference.com/es/difference-between-bfs-and-dfs-with-table/

https://www.encora.com/es/blog/dfs-vs-bfs