Proyecto de EDA 2: BFS y DFS

Ian Oñate, Marlon Zurita

2023-01-07

Tabla de Contenidos

1. Objetivos

- Implementar las funciones de búsqueda en anchura (BFS) y búsqueda en profundidad (DFS) para un grafo dirigido.
- Comprender la implementación de los algoritmos de DFS y BFS.

2. Introducción

Los algoritmos de búsqueda en profundidad (DFS) y búsqueda en anchura (BFS) son dos algoritmos fundamentales en la teoría de grafos. Estos algoritmos se utilizan para explorar un grafo, visitando todos los nodos del grafo en un orden específico.

Búsqueda en anchura (BFS)

El algoritmo BFS explora el grafo comenzando por un nodo inicial y explorando todos los vecinos de ese nodo, luego explora todos los vecinos de los vecinos del nodo inicial, y así sucesivamente. El algoritmo continúa explorando el grafo en esta forma hasta que ha visitado todos los nodos.

El algoritmo BFS se implementa típicamente utilizando una cola, ya sea de forma explícita o por medio de una función recursiva. La cola se utiliza para almacenar los nodos que aún no se han visitado. El algoritmo comienza agregando el nodo inicial a la cola. Luego, mientras la

cola no esté vacía, el algoritmo elimina el nodo primero de la cola y explora todos sus vecinos. Si un vecino no se ha visitado, el algoritmo lo agrega a la cola. El algoritmo se repite con este vecino hasta agotar todos los nodos del grafo (Geeks 2024).

Búsqueda en profundidad (DFS)

El algoritmo DFS explora el grafo comenzando por un nodo inicial y explorando todos los vecinos de ese nodo antes de pasar a otros nodos. DFS continúa explorando el grafo en esta forma hasta que ha visitado todos los nodos.

El algoritmo DFS se implementa típicamente utilizando una pila. La pila se utiliza para almacenar los nodos que aún no se han visitado. El algoritmo comienza agregando el nodo inicial a la pila. Luego, mientras la pila no esté vacía, el algoritmo elimina el nodo superior de la pila y explora todos sus vecinos. Si un vecino no se ha visitado, el algoritmo lo agrega a la pila (Geeks 2023).

Diferencias entre DFS y BFS

La principal diferencia entre DFS y BFS es el orden en el que exploran el grafo. El DFS explora el grafo en profundidad, mientras que el BFS explora el grafo en anchura.

Esta diferencia en el orden de exploración tiene implicaciones importantes para el rendimiento de los dos algoritmos. El DFS es más eficiente para encontrar el camino más corto entre dos nodos, mientras que el BFS es más eficiente para encontrar todos los componentes conectados de un grafo.

Aplicaciones de DFS y BFS

Los algoritmos de DFS y BFS se utilizan en diversas aplicaciones, por ejemplo:

- Encontrar caminos en un mapa
- Encontrar ciclos en un grafo
- Encontrar componentes conectados en un grafo
- Encontrar el camino más corto entre dos nodos

Los algoritmos de DFS y BFS son herramientas esenciales para cualquier programador que trabaje con grafos.

3. Desarrollo

```python
# Implementación del Grafo con ADT y OOP
from collections import deque

class Nodo:
    def __init__(self, dato):
        self.dato = dato
```

```
        self.vecinos = []

    def add_vecino(self, vecino):
        self.vecinos.append(vecino)

    def __str__(self):
        return f"{self.dato}: {', '.join(str(n) for n in self.vecinos)}"


class Arista:
    def __init__(self, fuente, dest, peso=None):
        self.fuente = fuente
        self.dest = dest
        self.peso = peso

    def __str__(self):
        return f"({self.fuente} -> {self.dest}, {self.peso})"


class Grafo:
    def __init__(self):
        self.nodos = {}

    def add_nodo(self, dato):
        if dato not in self.nodos:
            self.nodos[dato] = Nodo(dato)

    def add_arista(self, fuente, dest, peso=None):
        self.add_nodo(fuente)
        self.add_nodo(dest)
        self.nodos[fuente].add_vecino(Arista(fuente, dest, peso))

    def print_grafo(self):
        for nodo in self.nodos.values():
            print(nodo)

    def bfs(self, start):
        queue = deque([start])
        searched = set([start])
        while queue:
            current = queue.popleft()
```

```
            print(current.dato)
            for vecino in current.vecinos:
                if vecino.dest not in searched:
                    queue.append(self.nodos[vecino.dest])
                    searched.add(vecino.dest)

    def dfs(self, nodo, grafo, parent, searched, componentR):
        componentR.append(nodo.dato)
        searched[nodo.dato] = True
        print(f"{parent} -> {nodo}")
        print(f'Vecinos de {nodo}: {[vecino.dest for vecino in nodo.vecinos]}')

        for vecino in nodo.vecinos:
            if not searched[vecino.dest]:  # Access neighbor directly
                self.dfs(self.nodos[vecino.dest], grafo, nodo, searched, componentR)


# Para las ejecuciones se usa un grafo predefinido

grafo = Grafo()
grafo.add_nodo(0)
grafo.add_nodo(1)
grafo.add_nodo(2)
grafo.add_nodo(3)
grafo.add_nodo(4)
grafo.add_nodo(5)

grafo.add_arista(0,1)
grafo.add_arista(0,2)
grafo.add_arista(0,5)

grafo.add_arista(1,0)
grafo.add_arista(1,2)

grafo.add_arista(2,0)
grafo.add_arista(2,1)
grafo.add_arista(2,3)
grafo.add_arista(2,4)

grafo.add_arista(3,2)
grafo.add_arista(3,4)
```

```
grafo.add_arista(3,5)

grafo.add_arista(4,2)
grafo.add_arista(4,3)

grafo.add_arista(5,0)
grafo.add_arista(5,3)

# Demo BFS
print("BFS para el grafo actual:")
print()
grafo.bfs(grafo.nodos[0])
print()
# Demo DFS

nodoS = grafo.nodos[0]
searched = [False] * len(grafo.nodos)
parent = None
componentR = []
print("Recorrido del árbol DFS:")
print()
grafo.dfs(nodoS, grafo, parent, searched, componentR)
print()
print(f"La búsqueda DFS es: {componentR}")
```

```
BFS para el grafo actual:

0
1
2
5
0
3
4

Recorrido del árbol DFS:

None -> 0: (0 -> 1, None), (0 -> 2, None), (0 -> 5, None)
Vecinos de 0: (0 -> 1, None), (0 -> 2, None), (0 -> 5, None): [1, 2, 5]
0: (0 -> 1, None), (0 -> 2, None), (0 -> 5, None) -> 1: (1 -> 0, None), (1 -> 2, None)
Vecinos de 1: (1 -> 0, None), (1 -> 2, None): [0, 2]
1: (1 -> 0, None), (1 -> 2, None) -> 2: (2 -> 0, None), (2 -> 1, None), (2 -> 3, None), (2 -
```

```
Vecinos de 2: (2 -> 0, None), (2 -> 1, None), (2 -> 3, None), (2 -> 4, None): [0, 1, 3, 4]
2: (2 -> 0, None), (2 -> 1, None), (2 -> 3, None), (2 -> 4, None) -> 3: (3 -> 2, None), (3 ->
Vecinos de 3: (3 -> 2, None), (3 -> 4, None), (3 -> 5, None): [2, 4, 5]
3: (3 -> 2, None), (3 -> 4, None), (3 -> 5, None) -> 4: (4 -> 2, None), (4 -> 3, None)
Vecinos de 4: (4 -> 2, None), (4 -> 3, None): [2, 3]
3: (3 -> 2, None), (3 -> 4, None), (3 -> 5, None) -> 5: (5 -> 0, None), (5 -> 3, None)
Vecinos de 5: (5 -> 0, None), (5 -> 3, None): [0, 3]

La búsqueda DFS es: [0, 1, 2, 3, 4, 5]
```

4. Conclusiones

- El proyecto implementó correctamente los algoritmos BFS y DFS en un grafo dirigido. Los algoritmos obtuvieron el recorrido correcto en ambos casos, lo que valida su correcta adaptación al contexto del proyecto.
- La elección de estructuras de datos apropiadas, como colas para BFS y pilas para DFS, es la parte más importante de la implementación.
- El proyecto demuestra que los algoritmos BFS y DFS son herramientas poderosas que pueden ser utilizadas para resolver una variedad de problemas relacionados con grafos. En particular, recorrido de grafos, búsqueda de caminos y búsqueda de componentes conectados.

5. Referencias

Geeks, Geeks for. 2023. GeeksforGeeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.

———. 2024. GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.