Escuela Colombiana de Ingeniería Julio Garavito

Algoritmos y Estructuras de Datos 2024-1

Laboratorio GRAFOS

Andersson David Sánchez Méndez

Cristian Santiago Pedraza Rodríguez

David Eduardo Salamanca Aguilar

26 de abril de 2024

DOCUMENTO TÉCNICO

Requisitos

Especificación 1

Completar el método _getNeighborsMatAdj para que cuando el grafo dirigido(directed) sea False, mire la matriz de adyacencia para cada nodo donde halla 1, es decir, determinar los vecinos de cada nodo, pero teniendo en cuenta la matriz y no la lista de adyacencia, pero si se hace la comparación entre los dos, debería dar el mismo resultado. También, se puede ver el análisis de las búsquedas (por anchura y por profundidad).

Especificación 2

Implementar los métodos BFS y DFS para la matriz de adyacencia, con base al código ya completado de

_getNeighborsMatAdj.

Entrada general

La entrada es un número n que es el rango de las relaciones que habrá en el grafo, así el conjunto de relaciones es proporcional a ese número. Implícitamente las rutas serán generadas aleatoriamente teniendo en cuenta el n.

Salida general

La salida generará la lista y matriz de adyacencia, así como los métodos BFS y DFS aplicados al conjunto de relaciones (que en este caso son rutas porque está aplicado al prototipo funcional), y también los componentes conexa.

Diseño

Estrategia 1

- **Inicialización de la lista de vecinos:** Se crea una lista vacía neighbors que almacenará los vecinos del vértice dado.

- **Obtención del índice de la fila**: Se utiliza el diccionario encoder para obtener el índice de la fila correspondiente al vértice en la matriz de adyacencia. Este índice se almacena en la variable row.

- **Iteración a través de los elementos de la fila**: Se itera a través de cada elemento en la fila de la matriz de adyacencia correspondiente al vértice dado. Esta iteración se realiza utilizando un bucle for que recorre cada columna de la fila.

- **Consideración de conexiones según la dirección**: Dependiendo de si el grafo es dirigido o no dirigido, se decide cómo agregar los vecinos a la lista neighbors.

    - ✓ Si el grafo es dirigido, solo se consideran las conexiones desde el vértice dado hacia los vecinos. Por lo tanto, si la entrada en la matriz de adyacencia es 1 en la posición [row][col], se agrega el vértice decoder[col] a la lista de vecinos.
    - ✓ Si el grafo no es dirigido, se consideran todas las conexiones del vértice dado, independientemente de la dirección. Entonces, se agregan tanto los vecinos que tienen una conexión desde el vértice dado como los que tienen una conexión hacia él.

- Retorno de los vecinos encontrados: Se devuelve la lista neighbors, que contiene todos los vecinos del vértice dado.

Estrategia 2

- **Para BFS**:

  Se inicializan las propiedades de los vértices (_buildVProps) antes de comenzar el recorrido.

  Se itera sobre todos los nodos (source) y se verifica si el nodo no ha sido visitado (WHITE).

  Si el nodo no ha sido visitado, se ejecuta un recorrido BFS desde ese nodo. Dentro del recorrido BFS, se utiliza una cola (queue) para realizar el recorrido en anchura desde el nodo actual.

  Después de que la cola esté vacía, se marca el nodo actual como visitado (BLACK) y se continúa con el siguiente nodo no visitado.

- **Para DFS**:

  Se inicializan las propiedades de los vértices (_buildVProps) antes de comenzar el recorrido.

  Se itera sobre todos los nodos (source) y se verifica si el nodo no ha sido visitado (WHITE).

  Si el nodo no ha sido visitado, se ejecuta un recorrido DFS desde ese nodo. Dentro del recorrido DFS, se utiliza una función auxiliar (dfs_visit) para realizar el recorrido en profundidad desde el nodo actual.

  La función dfs_visit marca el nodo actual como visitado (GRAY) y luego recorre todos los vecinos del nodo actual. Si un vecino no ha sido visitado, se llama recursivamente a dfs_visit para visitar ese vecino.

Después de visitar todos los vecinos, se marca el nodo actual como completamente visitado (BLACK) y se actualiza el tiempo de finalización.

Casos de prueba

INPUT	OUTPUT
3	ADJ LIST, ADJ MATRIX, BFS, DFS, CONEXA
2	ADJ LIST, ADJ MATRIX, BFS, DFS, CONEXA
8	ADJ LIST, ADJ MATRIX, BFS, DFS, CONEXA
7	ADJ LIST, ADJ MATRIX, BFS, DFS, CONEXA
5	ADJ LIST, ADJ MATRIX, BFS, DFS, CONEXA

Para n = 3:

```
==================== ADJ List ======================
0 []
1 ['2']
2 []

==================== ADJ Matrix ======================
0 0 0
0 0 1
0 0 0

====================== BFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '1'}
2 --> {'color': 'black', 'distance': 1, 'parent': 1, 'path': '1-->2'}

====================== DFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 2, 'path': '0'}
1 --> {'color': 'black', 'distance': 3, 'parent': None, 'final': 6, 'path': '1'}
2 --> {'color': 'black', 'distance': 4, 'parent': 1, 'final': 5, 'path': '1-->2'}

====================== Connected components spread ==================
====================== Connected component found ==========================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '1'}
2 --> {'color': 'black', 'distance': 1, 'parent': 1, 'path': '1-->2'}
```

Para n = 2:

```
Para n = 2
==================== ADJ List ======================
0 ['1']
1 []

==================== ADJ Matrix ======================
0 1
0 0

==================== BFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}

==================== DFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 4, 'path': '0'}
1 --> {'color': 'black', 'distance': 2, 'parent': 0, 'final': 3, 'path': '0-->1'}

==================== Connected components spread ==================
==================== Connected component found ==========================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}
```

Para n = 8:

```
59  ==================== ADJ List ======================
60  0 ['6', '5']
61  1 ['5', '4', '3']
62  2 ['4', '3', '6', '5']
63  3 ['4']
64  4 ['6', '5', '7']
65  5 ['6']
66  6 []
67  7 []
68
69  ==================== ADJ Matrix ======================
70  0 0 0 0 0 1 1 0
71  0 0 0 1 1 1 0 0
72  0 0 0 1 1 1 1 0
73  0 0 0 0 1 0 0 0
74  0 0 0 0 0 1 1 1
75  0 0 0 0 0 0 1 0
76  0 0 0 0 0 0 0 0
77  0 0 0 0 0 0 0 0
78
```

```
====================== BFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->1'}
2 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->2'}
3 --> {'color': 'black', 'distance': 3, 'parent': 1, 'path': '0-->5-->1-->3'}
4 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->4'}
5 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->5'}
6 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->6'}
7 --> {'color': 'black', 'distance': 3, 'parent': 4, 'path': '0-->5-->4-->7'}

====================== DFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 16, 'path': '0'}
1 --> {'color': 'black', 'distance': 3, 'parent': 5, 'final': 14, 'path': '0-->5-->1'}
2 --> {'color': 'black', 'distance': 5, 'parent': 3, 'final': 12, 'path': '0-->5-->1-->3-->2'}
3 --> {'color': 'black', 'distance': 4, 'parent': 1, 'final': 13, 'path': '0-->5-->1-->3'}
4 --> {'color': 'black', 'distance': 6, 'parent': 2, 'final': 11, 'path': '0-->5-->1-->3-->2-->4'}
5 --> {'color': 'black', 'distance': 2, 'parent': 0, 'final': 15, 'path': '0-->5'}
6 --> {'color': 'black', 'distance': 7, 'parent': 4, 'final': 8, 'path': '0-->5-->1-->3-->2-->4-->6'}
7 --> {'color': 'black', 'distance': 9, 'parent': 4, 'final': 10, 'path': '0-->5-->1-->3-->2-->4-->7'}
```

```
====================== Connected components spread ==================
====================== Connected component found ============================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->1'}
2 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->2'}
3 --> {'color': 'black', 'distance': 3, 'parent': 1, 'path': '0-->5-->1-->3'}
4 --> {'color': 'black', 'distance': 2, 'parent': 5, 'path': '0-->5-->4'}
5 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->5'}
6 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->6'}
7 --> {'color': 'black', 'distance': 3, 'parent': 4, 'path': '0-->5-->4-->7'}
```

Para n = 7:

```
==================== ADJ List ======================
0 ['4', '3', '6']
1 ['6', '3']
2 ['4']
3 ['6', '5']
4 []
5 []
6 []

==================== ADJ Matrix ======================
0 0 0 1 1 0 1
0 0 0 1 0 0 1
0 0 0 0 1 0 0
0 0 0 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

====================== BFS ==================
==================== Results ======================
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 2, 'parent': 3, 'path': '0-->3-->1'}
2 --> {'color': 'black', 'distance': 2, 'parent': 4, 'path': '0-->4-->2'}
3 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->3'}
4 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->4'}
5 --> {'color': 'black', 'distance': 2, 'parent': 3, 'path': '0-->3-->5'}
6 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->6'}
```

```
143 ====================== DFS ==================
144 ===================== Results ======================
145 0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 14, 'path': '0'}
146 1 --> {'color': 'black', 'distance': 3, 'parent': 3, 'final': 6, 'path': '0-->3-->1'}
147 2 --> {'color': 'black', 'distance': 11, 'parent': 4, 'final': 12, 'path': '0-->4-->2'}
148 3 --> {'color': 'black', 'distance': 2, 'parent': 0, 'final': 9, 'path': '0-->3'}
149 4 --> {'color': 'black', 'distance': 10, 'parent': 0, 'final': 13, 'path': '0-->4'}
150 5 --> {'color': 'black', 'distance': 7, 'parent': 3, 'final': 8, 'path': '0-->3-->5'}
151 6 --> {'color': 'black', 'distance': 4, 'parent': 1, 'final': 5, 'path': '0-->3-->1-->6'}
152
153 ====================== Connected components spread ==================
154 ====================== Connected component found ==========================
155 ===================== Results ======================
156 0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
157 1 --> {'color': 'black', 'distance': 2, 'parent': 3, 'path': '0-->3-->1'}
158 2 --> {'color': 'black', 'distance': 2, 'parent': 4, 'path': '0-->4-->2'}
159 3 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->3'}
160 4 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->4'}
161 5 --> {'color': 'black', 'distance': 2, 'parent': 3, 'path': '0-->3-->5'}
162 6 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->6'}
163
```

Para n = 5:

```
166 ====================== ADJ List ======================
167 0 ['1', '2']
168 1 ['2', '3']
169 2 []
170 3 []
171 4 []
172
173 ====================== ADJ Matrix ======================
174 0 1 1 0 0
175 0 0 1 1 0
176 0 0 0 0 0
177 0 0 0 0 0
178 0 0 0 0 0
179
180 ====================== BFS ==================
181 ===================== Results ======================
182 0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
183 1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}
184 2 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->2'}
185 3 --> {'color': 'black', 'distance': 2, 'parent': 1, 'path': '0-->1-->3'}
186 4 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '4'}
187
```

```
188 ====================== DFS ==================
189 ====================== Results ======================
190 0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 8, 'path': '0'}
191 1 --> {'color': 'black', 'distance': 2, 'parent': 0, 'final': 7, 'path': '0-->1'}
192 2 --> {'color': 'black', 'distance': 3, 'parent': 1, 'final': 4, 'path': '0-->1-->2'}
193 3 --> {'color': 'black', 'distance': 5, 'parent': 1, 'final': 6, 'path': '0-->1-->3'}
194 4 --> {'color': 'black', 'distance': 9, 'parent': None, 'final': 10, 'path': '4'}
195
196 ====================== Connected components spread ==================
197 ====================== Connected component found ==========================
198 ====================== Results ======================
199 0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
200 1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}
201 2 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->2'}
202 3 --> {'color': 'black', 'distance': 2, 'parent': 1, 'path': '0-->1-->3'}
203 4 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '4'}
204
205
206
```

Análisis

Temporal 1

Al acceder a los elementos de una fila en la matriz de adyacencia, se toma tiempo constante, ya que los elementos de la fila son accesibles directamente mediante el índice de columna (un arreglo de arreglos). Por lo tanto, la complejidad para obtener los vecinos utilizando la matriz de adyacencia es O(1), tanto en el caso de grafos dirigidos como no dirigidos.

La complejidad de _getNeighborsMatAdj sería O(V), donde V es el número de vértices en el grafo. Esto se debe a que en el peor caso, se recorre una fila completa de la matriz de adyacencia, que tiene V elementos, para encontrar los vecinos del vértice dado.

Temporal 2

La complejidad de BFS (Breadth-First Search) es O(V + E), donde V es el número de vértices y E es el número de aristas en el grafo. En el peor caso, BFS visita cada vértice y cada arista del grafo una vez. En cada paso, BFS recorre todos los vértices adyacentes al vértice actual. Como máximo, esto implica revisar todas las aristas del grafo una vez. Por lo tanto, la complejidad de tiempo es proporcional al número total de vértices y aristas en el grafo.

Y por el otro lado, DFS (Depth-First Search) tiene la misma complejidad de tiempo que BFS, lo único que cambia es como se visitan los nodos, aquí DFS recorre uno de los vértices adyacentes al vértice actual y

continúa explorando en profundidad. Explora tan profundamente como sea posible antes de retroceder y explorar otros nodos.

Código 1

```python
def _getNeighborsMatAdj(self,vertex):
    neighbors = []
    row = self.encoder[vertex] # Obtiene el índice de la fila correspondiente al vértice en la matriz de adyacencia
    for col in range(len(self.adjMat[row])): # Itera a través de la fila correspondiente en la matriz de adyacencia
        if self.directed:
            # Si el grafo es dirigido, solo se consideran las conexiones desde el vértice dado hacia los vecinos.
            if self.adjMat[row][col] == 1:
                neighbors.append(self.decoder[col])
        else:
            # Si el grafo no es dirigido, se consideran todas las conexiones del vértice dado, independientemente de la dirección.
            if self.adjMat[row][col] == 1 or self.adjMat[col][row] == 1:
                neighbors.append(self.decoder[col])
    return neighbors
```

Código 2

```python
class Graph:

    2 usages  new *
    def bfs(self):
        self._buildVProps()
        for source in self.vertexes:
            if self.v_props[source]['color'] == WHITE:
                self.v_props[source]['color'] = GRAY
                self.v_props[source]['distance'] = 0
                self.v_props[source]['parent'] = None
                queue = [source]
                while queue:
                    u = queue.pop(0)
                    for neighbor in self._getNeighborsMatAdj(u):
                        if self.v_props[neighbor]['color'] == WHITE:
                            self.v_props[neighbor]['color'] = GRAY
                            self.v_props[neighbor]['distance'] = self.v_props[u]['distance'] + 1
                            self.v_props[neighbor]['parent'] = u
                            queue.append(neighbor)
                    self.v_props[u]['color'] = BLACK
        return self.v_props
```

```python
def dfs(self):
    self._buildVProps()
    time = 0
    for v in self.vertexes:
        if self.v_props[v]['color'] == WHITE:
            time = self.dfs_visit(v, time)
    return self.v_props

2 usages  new *
def dfs_visit(self, vertex, time):
    time = time + 1
    self.v_props[vertex]['distance'] = time
    self.v_props[vertex]['color'] = GRAY
    for neighbor in self._getNeighborsMatAdj(vertex):
        if self.v_props[neighbor]['color'] == WHITE:
            self.v_props[neighbor]['parent'] = vertex
            time = self.dfs_visit(neighbor, time)
    self.v_props[vertex]['color'] = BLACK
    time = time + 1
    self.v_props[vertex]['final'] = time
    return time
```

Documentación

Dentro del código.

Fuentes

Método insertado en el código compartido por el profesor sobre Grafos con el prototipo funcional de los métodos BFS y DFS de la matriz de adyacencia aplicado a un ejemplo de la vida cotidiana.

PROTOTIPO FUNCIONAL APLICADO A LA COTIDIANIDAD

Se presenta la aplicación de algoritmos de recorrido en un sistema de transporte público utilizando grafos. Se implementó los algoritmos de recorrido en anchura (BFS) y en profundidad (DFS) para encontrar rutas, determinar la conectividad entre paradas y resolver problemas relacionados con un sistema de autobuses.

Estructura del Grafo

El grafo utilizado en este sistema de transporte público está representado como un grafo no dirigido, donde los vértices representan las paradas de autobús y las aristas representan las rutas de autobús entre las paradas, es decir el directed = False. Es necesario ver como la lista y matriz de adyacencia se implementaron aquí:

- **Lista de Adyacencias:** Cada parada está asociada a una lista de las paradas a las que se puede llegar directamente desde ella.

-**Matriz de Adyacencias**: Una matriz booleana donde cada fila y columna representan una parada, y el valor en la posición (i, j) indica si hay una conexión directa entre las paradas i y j.

Algoritmos Implementados

BFS (Breadth-First Search):

El algoritmo BFS se utiliza para encontrar la ruta más corta entre dos paradas de autobús y determinar la conectividad entre todas las paradas.

DFS (Depth-First Search)

El algoritmo DFS se utiliza para encontrar la ruta entre dos paradas de autobús y también para encontrar componentes conectadas en el sistema de transporte.

Aplicación en un Sistema de Transporte Público

Se hicieron varias simulaciones partiendo del rango de las rutas que va a tomar el transporte público, que después con ese rango se convertirán en conjuntos, para construir el grafo; ya con eso se puede determinar la lista y matriz de adyacencia, los métodos BFS y DFS, y también para determinar los componentes conectados dentro de este prototipo.

Conclusiones

- Los resultados obtenidos al aplicar BFS y DFS en el sistema de transporte público muestran que todas las paradas están conectadas y que es posible encontrar rutas entre cualquier par de paradas.

- En la práctica, estos algoritmos son fundamentales en la planificación y optimización de redes de transporte público, ayudando a mejorar la eficiencia del sistema, reducir los tiempos de viaje y aumentar la satisfacción de los usuarios.