

Grafos dirigidos

¿Para qué sirven los grafos? Para representar relaciones arbitrarias entre objetos de datos.

Definiciones importantes:

Un grafo G es, esencialmente, un conjunto de vértices V y de arcos A .

Un arco está compuesto por un par de vértices, puede ser, por ejemplo (v, w) , donde uno se denomina "cola" y otro "cabeza". El arco (v, w) , donde "v" es la cola y "w" la cabeza, se puede representar: $v \rightarrow w$.

A partir de esta representación, se dice que "v va a w" y que "w es adyacente de v".

Los vértices, en un grafo dirigido, se pueden emplear para representar objetos, y los arcos, para establecer relaciones entre dichos objetos. Por ejemplo, los vértices representan ciudades, y los arcos, los vuelos que conectan dos ciudades.

Los vértices podrían representar asignaturas de una carrera universitaria, y los arcos las relaciones de previatura entre ellas.

Camino

Un camino en un grafo dirigido, es una secuencia de vértices V_n , tal que $V_1 \rightarrow V_2$, $V_2 \rightarrow V_3$, $V_3 \rightarrow V_4$, y así sucesivamente.

- . Un camino es **simple**, si todos sus vértices, menos tal vez, el primero y el último, son distintos.
- . Un **ciclo** es un camino, de al menos longitud 2, que empieza y termina en el mismo vértice.

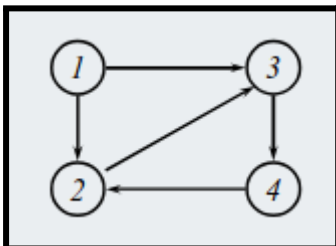
Longitud

Es la cantidad de arcos en un camino.

Se hace mención al caso especial, de un vértice único "v", el cual denota por sí mismo un camino de v a v con longitud 0.

Grafo conectado

Se dice que un grafo está conectado, si existe un camino entre cualquier par de vértices



Aquí, el sub-grafo, 2, 3, 4 está conectado.

Por cierto, eso es un grafo dirigido de 4 vértices y 5 aristas.

Formas de representar grafos dirigidos

Matriz de adyacencia

Requiere un espacio mínimo del orden de n^2 , siendo n la cantidad de vértices. Big $O(1)$.


Lista de adyacencias

Requiere una cantidad de espacio proporcional a la suma de la cantidad de arcos más la cantidad de vértices, Space complexity: $O(v + a)$ y Time complexity: Big $O(v)$ (equivalente a n), v es el número de vértices.

Las listas pueden representarse tanto en forma estática, como dinámica

TDA Grafo

TDA Grafo



- **Grafo (Vértices, Aristas)**
- Dado un **vértice origen**, indicar los **caminos mínimos** a todos los otros
- **Todos los caminos mínimos**, de todo vértice a todo otro
- **Centro de Grafo, excentricidad** de un vértice
- **Cerradura transitiva**
- **Búsqueda en profundidad** (recorrer sistemáticamente todo el grafo en profundidad)
- **Camino, Caminos**

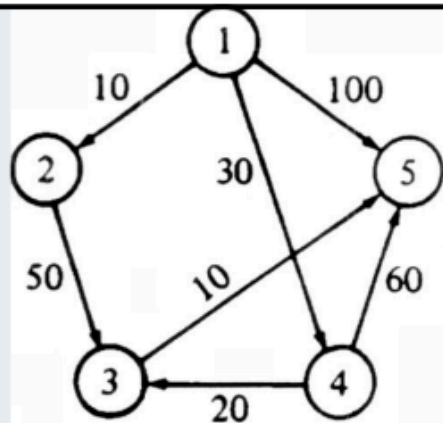
Algoritmos y Estructuras de Datos 12

Caminos más cortos con un origen

Algoritmo de Dijkstra

- Problema: Determinar el costo del camino más corto desde un origen hasta todos los otros vértices.
- Enfoque: Técnica “greedy”.
- Mantiene un conjunto S de vértices cuya distancia desde el origen ya se conoce.
- Inicialmente S contiene solo el vértice origen.
- En cada paso se agrega a S un vértice cuya distancia desde el origen es la más corta posible.
- Supuesto: Todas las aristas tienen costos no negativos.

```
1 Procedimiento Dijkstra(G, C, s):
2   Inicializar:
3     Para cada vértice v en V:
4       D[v] ← ∞           // Distancia inicial infinita
5       P[v] ← NULL       // Predecesor nulo
6     D[s] ← 0             // La distancia al origen es 0
7
8     Q ← V                // Cola de prioridad con todos los vértices
9
10    Mientras Q no esté vacío:
11      u ← vértice en Q con D[u] mínimo // Extraer nodo con distancia mínima
12      Eliminar u de Q
13
14      Para cada vecino v de u:         // Para cada arista (u, v)
15        Si v está en Q y D[u] + C[u][v] < D[v]:
16          D[v] ← D[u] + C[u][v]       // Relajación
17          P[v] ← u                    // Actualizar predecesor
18
19    Retornar D, P
```



Iteration	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
initial	{1}	—	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

Algoritmo de Floyd-Warshall

- Objetivo: Encontrar los caminos más cortos entre todos los pares de vértices.
- Complejidad: $O(n^3)$.
- Utiliza una matriz A donde se calculan las longitudes de los caminos más cortos.
- Fórmula: $A[i,j] = \min\{A[i,j], A[i,k] + A[k,j]\}$.
- Recuperación de caminos: Usando una matriz P donde $P[i,j]$ contiene el vértice k que permitió encontrar el valor más pequeño de $A[i,j]$. Si $P[i,j] = 0$, el camino más corto de i a j es directo.

```
1 Floyd(A: Array, C: Array) -> Array
2 COM
3   P <- Nuevo array
4
5   PARA CADA i DESDE 0 hasta N HACER
6     PARA CADA j DESDE 0 hasta N HACER
7       A[i,j] <- C[i,j]
8       P[i,j] <- 0
9     FIN PARA CADA j
10  FIN PARA CADA i
11
12  PARA CADA i DESDE 0 hasta N HACER
13    A[i,i] <- 0
14  FIN PARA CADA i
15
16  PARA CADA k DESDE 0 hasta N HACER
17    PARA CADA i DESDE 0 hasta N HACER
18      PARA CADA j DESDE 0 hasta N HACER
19        SI A[i,k] + A[k,j] < A[i,j] ENTONCES
20          A[i,j] <- A[i,k] + A[k,j]
21          P[i,j] <- k
22        FIN SI
23      FIN PARA CADA j
24    FIN PARA CADA i
25  FIN PARA CADA k
26
27  RETORNAR P
28 FIN
```

Excentricidad y Centro de un Grafo

- Excentricidad de un nodo v : La máxima distancia mínima entre v y cualquier otro nodo.
- Centro de G : El vértice con mínima excentricidad.
- Procedimiento para encontrar el centro:
 1. Aplicar Floyd-Warshall para obtener las longitudes de los caminos.
 2. Encontrar el máximo valor en cada columna (excentricidad e_i).
 3. Encontrar el vértice con excentricidad mínima (centro de G).

Transitividad y Caminos

- Dada una matriz C donde $C[i,j] = 1$ si hay una arista de i a j , se busca obtener una matriz A tal que $A[i,j] = 1$ si existe un camino de longitud ≥ 1 de i a j .

Búsqueda en profundidad (Depth-First Search)

- Generalización del recorrido preorden
- Grafo G , nodos inicialmente marcados como no visitados
- Selecciono vértice como inicio, se marca como visitado y empiezo a explorar desde ahí.
- Se recorre cada vértice no visitado adyacente a v , usando BPF recursivamente
- Se termina cuando todos los nodos se pueden alcanzar desde v . Si quedan vértices por visitar, se selecciona otro vértice como punto de partida y se repite la búsqueda.

En Java:

```
1 public Collection<TVertice> bpf(Comparable origen) {
2     Set<Comparable> visitados= new HashSet<>();
3     bpfRecursoivo(origen, visitados);
4     return visitados.stream().map(v ->
5         (TVertice)buscarVertice(v)).collect(Collectors.toList());
6 }
7
8 private void bpfRecursoivo(Comparable actual, Set<Comparable> visitados) {
9     visitados.add(actual);
10    IVertice verticeActual = vertices.get(actual);
11    if (verticeActual!= null) {
12        for (Object o: verticeActual.getAdyacentes()) {
13            TAdyacencia adyacente = (TAdyacencia) o;
14            Comparable destino= adyacente.getDestino().getEtiqueta();
15            if (!visitados.contains(destino)) {
16                bpfRecursoivo(destino, visitados);
17            }
18        }
19    }
20 }
```

