Algoritmus Et Structex
Compendiem Grafex Dirigidum (UT7)

Santiago Blanco

12-06-2025

Grafos Dirigidos

- Un grafo dirigido G consiste en un conjunto de vértices V y un conjunto de aristas A.
- Una arista es un par ordenado de vértices (v,w) donde v es la cola y w la cabeza (representado como $v \longrightarrow w$).
- w es adyacente a v.

Caminos

- **Camino**: Secuencia de vértices $v_1, v_2, v_3, \ldots$ tal que $v_1 \longrightarrow v_2$, $v_2 \longrightarrow v_3, \ldots$ son aristas.
- La longitud de un camino es el número de aristas que lo componen.
- **Camino simple**: Todos sus vértices, excepto posiblemente el primero y el último, son distintos.
- **Ciclo simple**: Camino simple de longitud ≥1 que comienza y termina en el mismo vértice (también llamados bucles).

Representación de Grafos

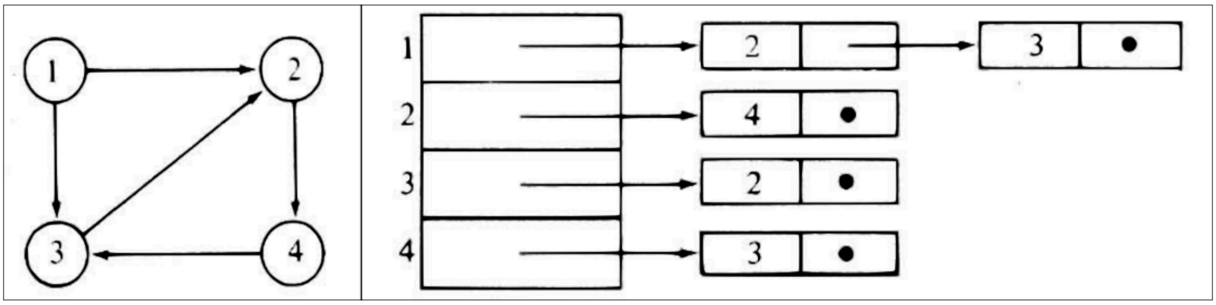
Matriz de Adyacencia

- Para un grafo G = (V,A), su matriz de adyacencia es una matriz A de dimensión n×n con valores booleanos, donde A[i,j] es verdadero si existe un arco del vértice i al j.
- Alternativamente, puede contener costos en vez de valores booleanos, donde A[i,j] es el costo de la arista desde i hasta j.
- **Problema**: Requiere espacio $\Omega(n^2)$ independientemente de la cantidad de aristas k.
- Complejidad: $O(n^2)$ para leer la matriz.

	1	2	3	4
1		*a*		*b*
2	*a*		*b*	
3		*b*		*a*
4	*b*		*a*	

Listas de Adyacencia

- La lista de adyacencia para un vértice v es una lista de todos los vértices adyacentes a v.
- **Ventaja**: Almacenamiento proporcional a la suma de la cantidad de vértices y aristas.
- **Desventaja**: O(n) para determinar si hay una arista entre dos vértices.



Algoritmos para Caminos más Cortos

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.

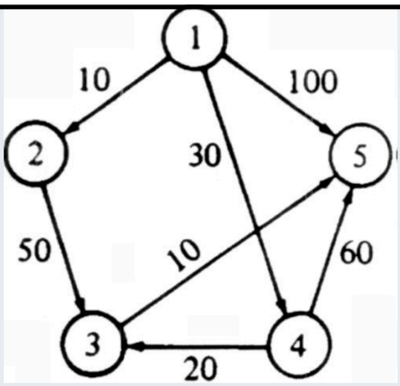
```
Procedimiento Dijkstra(G, C, s):
    Inicializar:
        Para cada vértice v en V:
            D[v] ← ∞          // Distancia inicial infinita
            P[v] ← NULL       // Predecesor nulo
        D[s] ← 0              // La distancia al origen es 0

        Q ← V                 // Cola de prioridad con todos los vértices

    Mientras Q no esté vacío:
        u ← vértice en Q con D[u] mínimo  // Extraer nodo con distancia mínima
        Eliminar u de Q

        Para cada vecino v de u:          // Para cada arista (u, v)
            Si v está en Q y D[u] + C[u][v] < D[v]:
                D[v] ← D[u] + C[u][v]    // Relajación
                P[v] ← u                 // Actualizar predecesor

    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**: $A_k[i,j] = \min\{A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]\}$.
- **Recuperación de caminos**: Usando una matriz P donde P[i,j] contiene el vértice k que permitió encontrar el valor más pequeño de $A_k[i,j]$. Si P[i,j] = 0, el camino más corto de i a j es directo.

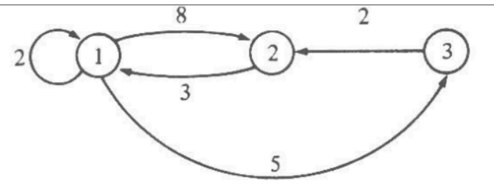
```
Floyd(A: Array, C: Array) -> Array
COM
  P <- Nuevo array

  PARA CADA i DESDE 0 hasta N HACER
    PARA CADA j DESDE 0 hasta N HACER
      A[i,j] <- C[i,j]
      P[i,j] <- 0
    FIN PARA CADA
  FIN PARA CADA

  PARA CADA i DESDE 0 hasta N HACER
    A[i,i] <- 0
  FIN PARA CADA

  PARA CADA k DESDE 0 hasta N HACER
    PARA CADA i DESDE 0 hasta N HACER
      PARA CADA j DESDE 0 hasta N HACER
        SI A[i,k] + A[k,j] < A[i,j] ENTONCES
          A[i,j] <- A[i,k] + A[k,j]
          P[i,j] <- k
        FIN SI
      FIN PARA CADA
    FIN PARA CADA
  FIN PARA CADA

  RETORNAR P
FIN
```



	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

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 i (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 busqueda.

En Java:

```java
public Collection<TVertice> bpf(Comparable origen) {
        Set<Comparable> visitados= new HashSet<>();
        bpfRecursivo(origen, visitados);
        return visitados.stream().map(v ->
        (TVertice)buscarVertice(v)).collect(Collectors.toList());
    }

private void bpfRecursivo(Comparable actual, Set<Comparable> visitados) {
        visitados.add(actual);
        IVertice verticeActual = vertices.get(actual);
        if (verticeActual!= null) {
            for (Object o: verticeActual.getAdyacentes()) {
                TAdyacencia adyacente = (TAdyacencia) o;
                Comparable destino= adyacente.getDestino().getEtiqueta();
                if (!visitados.contains(destino)) {
                    bpfRecursivo(destino, visitados);
                }
            }
        }
    }
```

En pseudocódigo

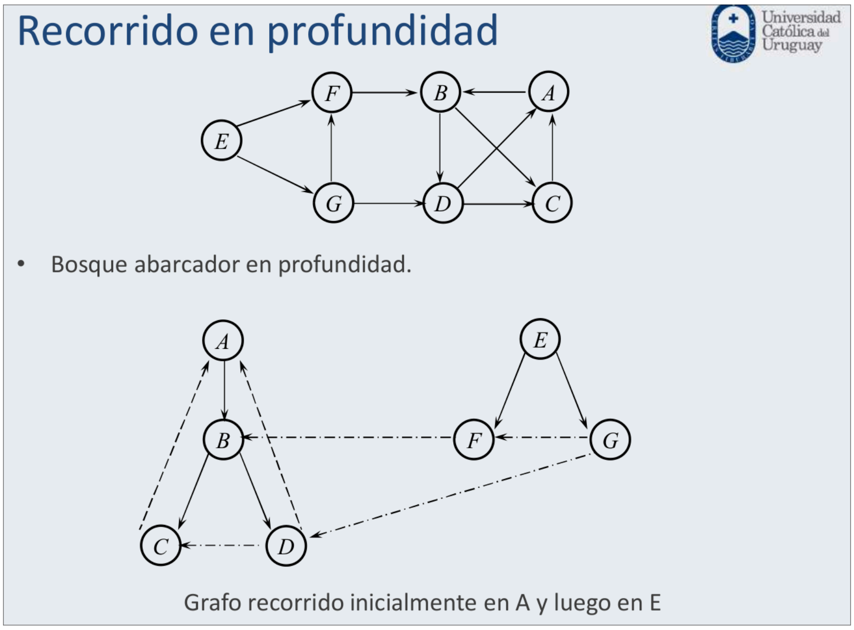
```
bpf(origen: Comparable) -> Collection<TVertice>
COM
  visitados <- conjunto vacío de Comparable
  bpfRecursivo(origen, visitados)
  RETORNAR lista de vértices correspondiente a las etiquetas en visitados
FIN

```

```
bpfRecursivo(actual: Comparable, visitados: Conjunto<Comparable>)
COM
  agregar actual a visitados
  verticeActual <- vertices.obtener(actual)

  SI verticeActual ≠ nulo ENTONCES
    PARA CADA adyacencia EN verticeActual.getAdyacentes() HACER
      destino <- adyacencia.getDestino().getEtiqueta()
      SI destino NO está en visitados ENTONCES
        bpfRecursivo(destino, visitados)
      FIN SI
    FIN PARA
  FIN SI
FIN
```

Bosque abarcador en profundidad



Los arcos que llevan a vértices nuevos se conocen como **arcos de árbol** y forman un "bosque abarcador en profundidad".

Existen otros tres tipos de arcos:

- **Arco de retroceso:** Va de un vértice a uno de sus antecesores en el árbol.
- **Arco de avance:** Va de un vértice a un descendiente propio.
- **Arco cruzado:** Va de un vértice a otro que no es ni antecesor ni descendiente.

Obtención de caminos

Este algoritmo, toma como parámetro el vértice de inicio y el destino. Recorre los vértices del grafo de manera recursiva, pasando por cada vértice adyacente de un vértice (w). En caso de que de que el vértice adyacente no sea el destino, se suma el camino hasta ese vértice adyacente al camino actual, en caso de que dicho vértice sí sea el nodo destino, guarda el valor del camino hasta dicho vértice. Para que esto se lleve a cabo, cada nodo tiene un atributo bool que verifica si ya fue visitado o no para que no se visite el mismo nodo 2 veces. Al final el nodo por el que se empezó del camino completo.

```
obtenerCamino(destino: TVertice, elCamino: TCamino, losCaminos: TCaminos)
COM
  this.visitado <- VERDADERO // O(1)
  elCamino.agregar(this) // O(1)

  SI w == destino ENTONCES
      losCaminos.agregar(elCamino + destino) // O(1)
  FIN SI

  PARA CADA ady EN adyacentes HACER // O(n^2)
    w <- ady.destino // O(1)
    SI w.visitado() == FALSO ENTONCES
      obtenerCamino(destino, elCamino, losCaminos)
    FIN SI
  FIN PARA CADA

  elCamino.quitar(this) // O(1)
  this.visitado <- FALSO // O(1)
FIN
```

Java:

```java
public TCaminos todosLosCaminos(Comparable etVertDest, TCamino caminoPrevio, TCaminos todosLosCaminos) {
    this.setVisitado(true);
    for (TAdyacencia adyacencia : this.getAdyacentes()) {
        TVertice destino = adyacencia.getDestino();
        if (!destino.getVisitado()) {
            if (destino.getEtiqueta().compareTo(etVertDest) == 0) {
                TCamino copia = caminoPrevio.copiar();
                copia.agregarAdyacencia(adyacencia);
                todosLosCaminos.getCaminos().add(copia);
            } else {
                TCamino copia = caminoPrevio.copiar();
                copia.agregarAdyacencia(adyacencia);
                destino.todosLosCaminos(etVertDest, copia, todosLosCaminos);
            }
        }
    }
        this.setVisitado(false);
        return todosLosCaminos;
}
```

Grafos dirigidos acíclicos

- El GDA es un grafo dirigido sin ciclos.
- Son más generales que los árboles, pero menos que los grafos dirigidos arbitrarios.
- Útiles para representar expresiones aritméticas con subexpresiones comunes.
- También son apropiados para representar órdenes parciales.

Prueba de aciclicidad

Se realiza búsqueda en profundidad y si se encuentra un arco de retroceso, el grafo tiene un ciclo.

- Si un grafo dirigido tiene un ciclo, siempre habrá un arco de retroceso en la búsqueda en profundidad.

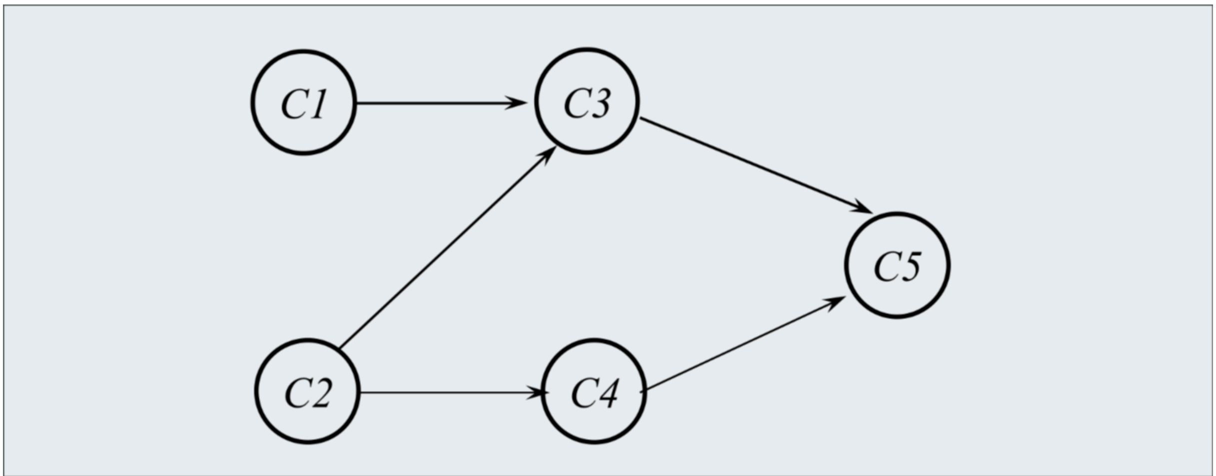
```java
public boolean tieneCiclo() {
  desvisitarVertices();
  boolean res = false;

  for (IVertice vertV : vertices.values()) {
    if (!vertV.getVisitado()) {
      LinkedList camino = new LinkedList();
      camino.add(vertV.getEtiqueta());
      res = ((TVertice) vertV).tieneCiclo(camino);
      if (res) {
        return true;
      }
    }
  }
  return res;
}
```

Clasificación topológica

Es un ordenamiento de los nodos en un GDA donde, para cada arista dirigida de i a j, i aparece antes que j.

Ej: Previatura de cursos.



- Clasificación topológica del grafo: C1, C2, C3, C4, C5
- Se pueden invertir las aristas para indicar dependencias. C5 depende de C3 y C4. Ejecutar BPF
- **Un grafo con ciclo no tiene Clasificación topológica**

```
procedure ClasificacionTopologica ();
w : Tvertice;
w : Tvertice;
COM
  (1) Visitar();
  (2) Para cada adyacente w hacer
  (3)
  Si no(w.visitado()) entonces
    w.ClasificacionTopologica()
  Fin Si
  Fin para cada
  imprimir (); //agregar “this” al principio de la lista de previas….
FIN; {ClasificacionTopologica}
```

Topological sort

1. Elegir un nodo no visitado
2. Ejecutar BPF explorando sólo nodos no visitados
3. En cada llamada recursiva, agrego el nodo actual a una lista, al final quedan en orden reverso.