Grafos (1)

Introducción

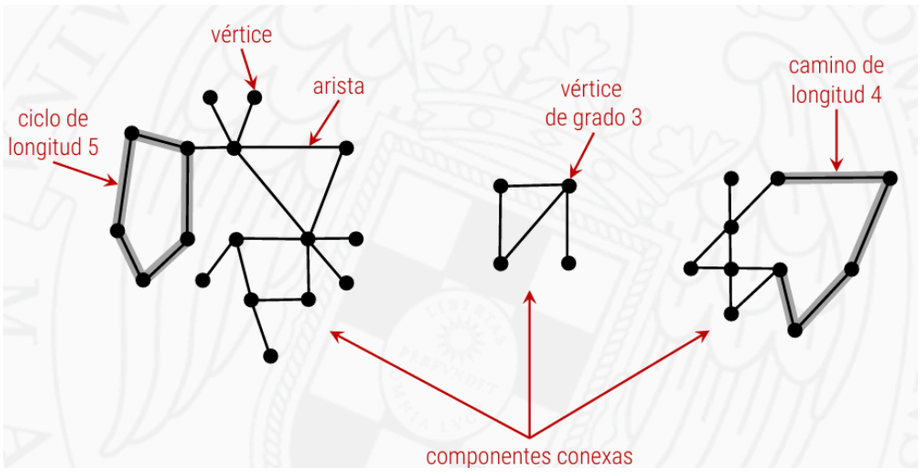
Los grafos sirven para representar elementos y conexiones uno a uno entre ellos

Aplicación	Elemento	Conexión
mapa	intersección	calle, carretera
internet	subred clase C	cable de red
web	página	enlace
red social	persona	amistad
juego	estado del tablero	movimiento legal
circuito	puerta lógica, transistor	cable
red de metro	estación	vía

Grafos No Dirigidos

Es un conjunto de **vértices** conectadas por **aristas**.

- Grado → número de conexiones de un vértice
- Camino → conexión entre un vértice a otro conectado donde la longitud es el número de aristas
- Ciclo → camino donde su vértice inicial es el final también



💡 Los nombres de los grafos no son importantes por los que numeramos de 0 a V-1. Si los nombres fuesen necesarios se realizaría una tabla de símbolos

Problemas

problema	descripción
caminos s-t	existe camino entre s y t
camino más corto s-t	camino mas corto entre s y t
grafo conexo	existe un camino entre todo par de vértices
ciclo	existe un ciclo en el grafo

problema	descripción
ciclo euleriano	existe un ciclo que utiliza cada arista del grafo exactamente una vez
ciclo hamiltoniano	existe un ciclo que pas cada vértice del grafo exactamente una vez
grafo bipartito	se pueden repartir vértices en dos conjuntos de tal forma que las aristas siempre conecten vértices en conjuntos distintos
grafo planar	Puede dibujarse el grafo en un plano sin que haya aristas que se crucen
grafos isomorfos	Existe un isomorfismo entre los grafos

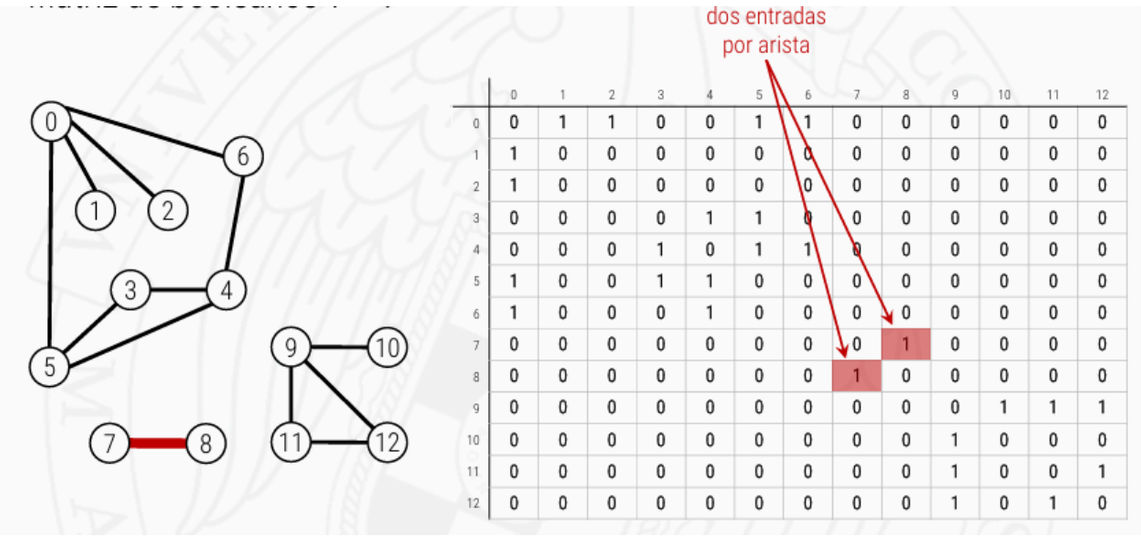
TAD

El TAD de los grafos cuenta con las siguientes operaciones:

- crear un grafo vacío, `Grafo(int V)`
- añadir una arista, `void ponArista(int v, int w)`
- consultar los adyacentes a un vértice, `Adys ady(int v) const`
- consultar el número de vértices, `int V() const`
- consultar el número de aristas, `int A() const`

Los grafos se pueden representar de diferentes formas:

Matriz de Adyacencia



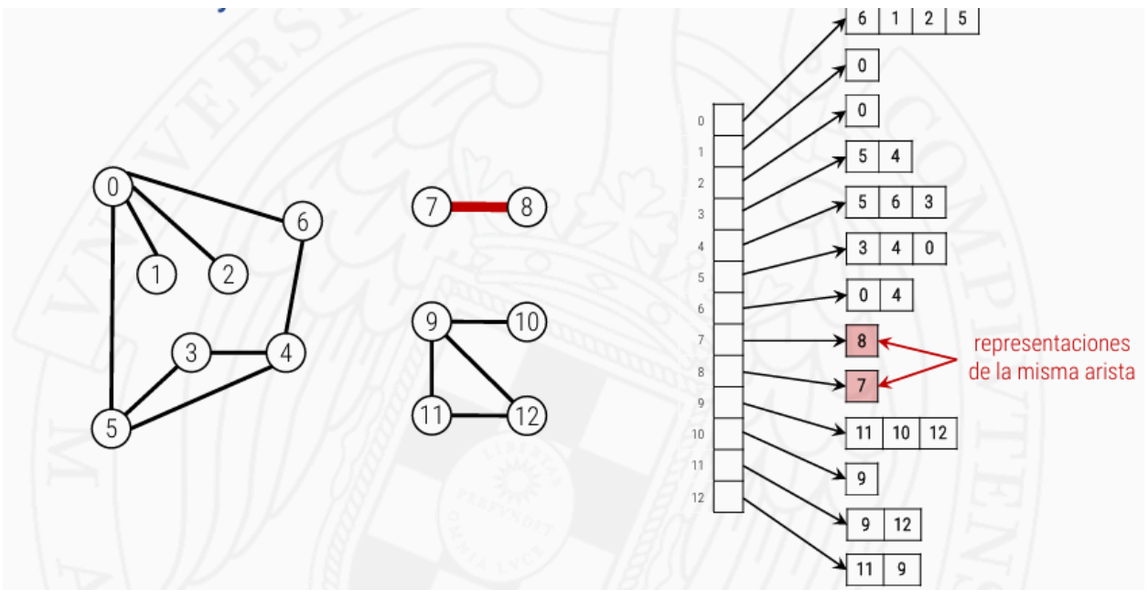
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Para las funciones (por ejemplo) grado y aristas el coste seria V y V^2

```
int grado(Grafo const& g, int v) { //O(V)
 int grado = 0;
 for (int w : g.ady(v))
    ++grado;
 return grado;
}
```

```
int aristas(Grafo const& g) { //O(V^2)
 int aristas = 0;
 for (int v = 0; v < g.V(); ++v)
    aristas += grado(g, v);
 return aristas / 2;
}
```

Lista de Adyacentes



Para las anteriores funciones seria grado(v) y O(V+A)

```
int grado(Grafo const& g, int v) { //O(grado(v))
 int grado = 0;
 for (int w : g.ady(v))
    ++grado;
 return grado;
}

int aristas(Grafo const& g) { //O(V + A)
 int aristas = 0;
 for (int v = 0; v < g.V(); ++v)
    aristas += grado(g, v);
 return aristas / 2;
}
```

Representación Elegida

💡 En la mayoría de los grados son dispersos y no densos. Es decir que el número de aristas esta más cerca de V que de V^2.
El número máximo de aristas es V*(V-1)/2

representacion	espacio	añadir arista v - w	comprobar si v y w son adyacentes	recorrer los vértices adyacentes a v
matriz de adyacencia	V^2	1	1	V
listas de adyacentes	V + A	1	grado(v)	grado(v)
conjuntos de adyacentes	V + A	log(V)	logV	grado(v)
listas de aristas	A	1	A	A

Implementación

```
using Adys = std::vector<int>; // lista de adyacentes a un vértice
class Grafo {
private:
    int _V; // número de vértices
    int _A; // número de aristas
    std::vector<Adys> _ady; // vector de listas de adyacentes
public:
    Grafo(int V) : _V(V), _A(0), _ady(_V) {}
    int V() const { return _V; }
    int A() const { return _A; }

 void ponArista(int v, int w) {
    if (v < 0 || v >= _V || w < 0 || w >= _V)
          throw std::domain_error("Vertice inexistente");
    ++_A;
    _ady[v].push_back(w);
    _ady[w].push_back(v);
 }
 Adys const& ady(int v) const {
    if (v < 0 || v >= _V)
       throw std::domain_error("Vertice inexistente");
    return _ady[v];
 }
};
```

Patrón de Diseño para el Procesamiento de Grafos

- Objetivo → separar la resolución de un problema de la representación del grafo
- Para cada problema sobre grafos que resolvamos crearemos una clase específica, `Problema`
- Generalmente, el constructor realizará cierto trabajo sobre el grafo y creará estructuras para contestar eficientemente a las preguntas del problema
- El usuario creará un grafo, después creará un objeto de la clase `Problema` pasándole el grafo como argumento a la constructora, y por último utilizará los métodos de consulta de esta clase para averiguar propiedades del grafo.

Ejemplo

Dado un grafo y un vértice origen s, determinar con qué otros vértices está conectado s.

```
class Conexion {
public:
    Conexion(Grafo const& g, int s); // busca vértices conectados a s
    bool conectado(int v) const; // ¿está v conectado a s?
    int cuantos() const; // ¿cuántos vértices están conectados a s?
};

void resuelve(Grafo const& g, int s) {
    Conexion conex(g,s);
    cout << "Vértices conectados a " << s << ":";
    for (int v = 0; v < g.V(); ++v) {
        if (v != s && conex.conectado(v))
            cout << ' ' << v;
    }
    cout << '\n';

    if (conex.cuantos() != g.V()) cout << "no ";
    cout << ”es conexo\n";
}
```

Recorrido en Profundidad

El recorrido o búsqueda en profundidad (*depth-first search*) imita la resolución de un laberinto:

- Para recorrer un grafo utilizaremos un algoritmo recursivo que va visitando vértices
- Visitar un vértice v consiste en:
    1. Marcarlo como visitado.

2. Hacer algo con él.
3. Visitar (recursivamente) todos los vértices adyacentes a v aún no visitados.

- El coste está en O(V + A)

Caminos desde Origen (ejemplo)

```
class CaminosDFS {
private:
    std::vector<bool> visit; // visit[v] = ¿hay camino de s a v?
    std::vector<int> ant; // ant[v] = último vértice antes de llegar a v
    int s; // vértice origen
    void dfs(Grafo const& G, int v) {
        visit[v] = true;
        for (int w : G.ady(v)) {
            if (!visit[w]) {
                ant[w] = v;
                dfs(G, w);
            }
        }
    }
public:
    CaminosDFS(Grafo const& g, int s) : visit(g.V(), false), ant(g.V()), s(s) {
        dfs(g, s);
    }
    // ¿hay camino del origen a v?
    bool hayCamino(int v) const {
        return visit[v];
    }

    using Camino = std::deque<int>; // para representar caminos
    // devuelve un camino desde el origen a v (debe existir)
    Camino camino(int v) const {
        if (!hayCamino(v))
            throw std::domain_error("No existe camino");
        Camino cam;
        // recuperamos el camino retrocediendo
        for (int x = v; x != s; x = ant[x])
            cam.push_front(x);
        cam.push_front(s);
        return cam;
    }
};
```

Recorrido en Anchura

En ocasiones buscamos encontrar el camino más corto desde un origen s a otro vértice v (o a todos los vértices conectados a s). El **recorrido en anchura** o breadth-first search lo logra. Los pasos son:

1. Visita todos los vértices alcanzables siguiendo una arista
2. Visita todos los vértices alcanzable utilizando dos aristas
3. Y así sucesivamente

Para lograrlo utiliza una cola donde guardar los vértices alcanzados pero que aún no se han explorado sus adyacentes.

El coste tambien será O(V+A)

Caminos más Cortos desde Origen

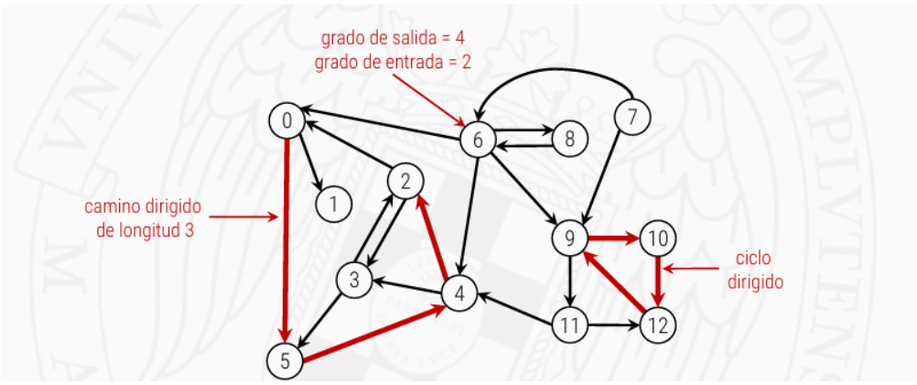
```
class CaminoMasCorto {
public:
    CaminoMasCorto(Grafo const& g, int s) : visit(g.V(), false), ant(g.V()), dist(g.V
        bfs(g);
    }
    // ¿hay camino del origen a v?
    bool hayCamino(int v) const {
        return visit[v];
    }

    // número de aristas entre s y v
    int distancia(int v) const {
        return dist[v];
    }
    // devuelve el camino más corto desde el origen a v (si existe)
    Camino camino(int v) const {
        if (!hayCamino(v)) throw std::domain_error("No existe camino");
        Camino cam;
        for (int x = v; x != s; x = ant[x])
            cam.push_front(x);
        cam.push_front(s);
        return cam;
    }
private:
    std::vector<bool> visit; // visit[v] = ¿hay camino de s a v?
    std::vector<int> ant; // ant[v]  = último vértice antes de llegar a v
    std::vector<int> dist; // dist[v] = aristas en el camino s-v más corto
    int s;

    void bfs(Grafo const& g) {
        std::queue<int> q;
        dist[s] = 0; visit[s] = true;
        q.push(s);
        while (!q.empty()) {
```

```
            int v = q.front(); q.pop();
            for (int w : g.ady(v)) {
               if (!visit[w]) {
                     ant[w] = v; dist[w] = dist[v] + 1; visit[w] = true;
                     q.push(w);
                  }
               }
            }
         }
      };
```

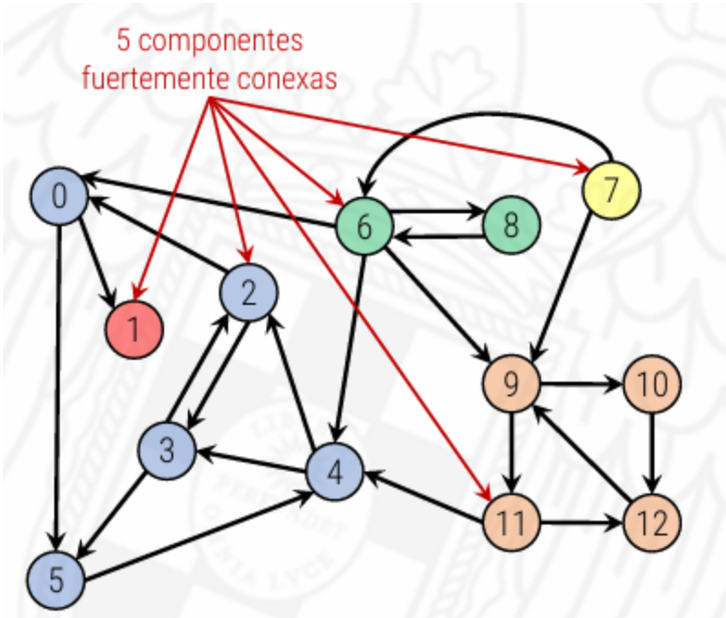
Grafos Dirigidos

También **digrafo**. Es un conjunto de vértices y un conjunto de *aristas dirigidas* (pares ordenados de vértices).

- Grado de salida → num de aristas que salen de un nodo
- Grado de entrada → num de aristas que entra a un nodo



Una componente fuertemente conexas son sub-grafos aislados o aquellos en los que cada par de vertices existen dos caminos unos que une el primer vértice con el segundo y otro que une el segundo con el primero.



Aplicaciones

aplicación	elemento	conexión
mapa	intersección	calle de sentido único
planificación	tarea	precedencia
web	página	enlace
referencias	artículo	cita
juego	estado del tablero	movimiento legal
memoria dinámica	objeto	punter
orientación a objetos	clase	herencia

Problemas

problem	descripción
camino s → t	existe un camino dirigido de s a t
camino más corto s → t	cual es el camino dirigido más corto (menos aristas) desde s hasta t
ciclo dirigido	existe un ciclo dirigido en el grafo
grafo fuertemente conexo	existe un camino dirigido entre todo par de vértices
ordenación topológica	Pueden los vértices ordenarse de forma que todas las aristas apunten en el mismo sentido
cierre transitivo	Para qué vértices v y w existe un camino v → w
PageRank	Cual es la importancia de una pagina web

TAD

El TAD de los grafos dirigidos cuenta con las siguientes operaciones:

- crear un grafo vacío, `Digrafo(int V)`
- añadir una arista, `void ponArista(int v, int w)`
- consultar los adyacentes a un vértice, `Adys ady(int v) const`
- consultar el número de vértices, `int V() const`
- consultar el número de aristas, `int A() const`
- calcular el grafo inverso, `Digrafo inverso() const`

Al igual que en los grafos, se pueden representar con matriz de adyacencia, donde cada entrada es una arista o por una lista de adyacentes.

En general se utilizara la lista de adyacentes para algoritmos basados en recorrer los adyacentes a un vértice.

representación	espacio	añadir arista v → w	comprobar si v y w son adyacentes	recorre los vértices adyacentes a v
matriz de adyacencia	V^2	1	1	V
lista de adyacentes	V+A	1	grado-sal(v)	grado-sal(v)
lista de aristas	A	1	A	A

```
using Adys = std::vector<int>; // lista de adyacentes a un vértice
class Digrafo {
private:
    int _V; // número de vértices
    int _A; // número de aristas
    std::vector<Adys> _ady; // vector de listas de adyacentes
public:
    Digrafo(int V) : _V(V), _A(0), _ady(_V) {}
    int V() const { return _V; }
    int A() const { return _A; }

    void ponArista(int v, int w) {
        if (v < 0 || v >= _V || w < 0 || w >= _V)
            throw std::domain_error("Vertice inexistente");
        ++_A;
        _ady[v].push_back(w);
    }

    Adys const& ady(int v) const {
        if (v < 0 || v >= _V)
            throw std::domain_error("Vertice inexistente");
        return _ady[v];
    }

   Digrafo inverso() const {
        Digrafo inv(_V);
        for (int v = 0; v < _V; ++v) {
            for (int w : _ady[v]) {
                inv.ponArista(w, v);
            }
        }
        return inv;
    }
};
```

Recorrido en Profundidad

```
class DFSDirigido {
public:
    DFSDirigido(Digrafo const& g, int s) : visit(g.V(), false) {
        dfs(g, s);
    }
    bool alcanzable(int v) const {
        return visit[v];
    }
private:
    std::vector<bool> visit; // visit[v] = ¿hay camino dirigido de s a v?

    void dfs(Digrafo const& g, int v) {
        visit[v] = true;
        for (int w : g.ady(v))
            if (!visit[w]) dfs(g, w);
    }
};
```

Recorrido en Anchura

```
 class BFSDirigido {
 public:
    BFSDirigido(Digrafo const& g, int s) : visit(g.V(), false), ant(g.V()), dist(g.V()), s
          bfs(g);
    }

    bool hayCamino(int v) const {
        return visit[v];
    }

    int distancia(int v) const {
        return dist[v];
    }
    Camino camino(int v) const {
        if (!hayCamino(v)) throw std::domain_error("No existe camino");
        Camino cam;
        for (int x = v; x != s; x = ant[x])
            cam.push_front(x);
        cam.push_front(s);
        return cam;
    }
private:
 std::vector<bool> visit; // visit[v] = ¿hay camino de s a v?
 std::vector<int> ant; // ant[v]  = último vértice antes de llegar a v
```

```
 std::vector<int> dist; // dist[v] = aristas en el camino s→v más corto
 int s;
 void bfs(Digrafo const& g) {
    std::queue<int> q;
    dist[s] = 0; visit[s] = true;
    q.push(s);
    while (!q.empty()) {
       int v = q.front(); q.pop();
       for (int w : g.ady(v)) {
          if (!visit[w]) {
             ant[w] = v; dist[w] = dist[v] + 1; visit[w] = true;
             q.push(w);
          }
       }
    }
 }
};
```

Ciclos Dirigidos

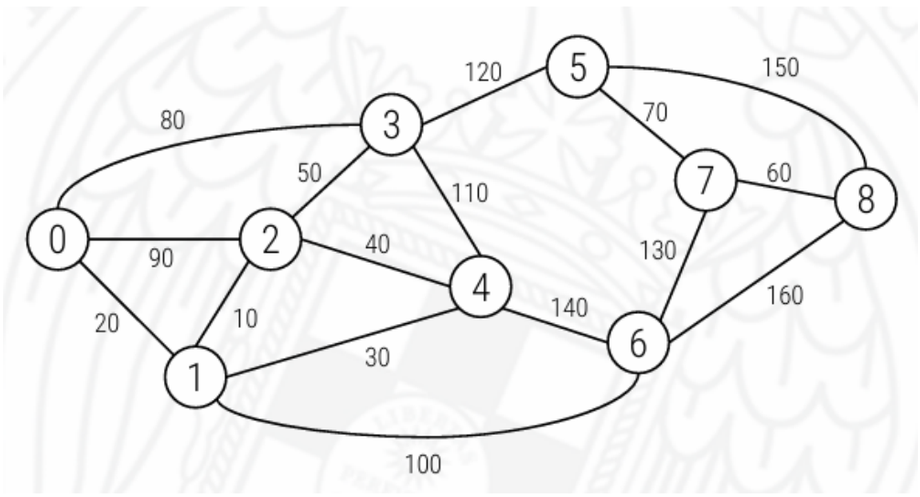
Detección de ciclos

```
 class CicloDirigido {
 public:
    CicloDirigido(Digrafo const& g) : visit(g.V(),false), ant(g.V()), apilado(g.V(),fals
       for (int v = 0; v < g.V(); ++v)
          if (!visit[v])
             dfs(g, v);
    }
    bool hayCiclo() const { return hayciclo; }
    Camino const& ciclo() const { return _ciclo; }
private:
     std::vector<bool> visit; // visit[v] = ¿se ha alcanzado a v en el dfs?
       std::vector<int>  ant; // ant[v] = vértice anterior en el camino a v
       std::vector<bool> apilado; // apilado[v] = ¿está el vértice v en la pila?
       Camino _ciclo; // ciclo dirigido (vacío si no existe)
       bool hayciclo;

       void dfs(Digrafo const& g, int v) {
          apilado[v] = true;
          visit[v] = true;
          for (int w : g.ady(v)) {
             if (hayciclo) // si hemos encontrado un ciclo terminamos
                return;
             if (!visit[w]) { // encontrado un nuevo vértice, seguimos
                ant[w] = v; dfs(g, w);
             } else if (apilado[w]) { // hemos detectado un ciclo
```

```
                // se recupera retrocediendo
                hayciclo = true;
                for (int x = v; x != w; x = ant[x])
                    _ciclo.push_front(x);
                _ciclo.push_front(w); _ciclo.push_front(v);
               }
            }
            apilado[v] = false;
        }
```

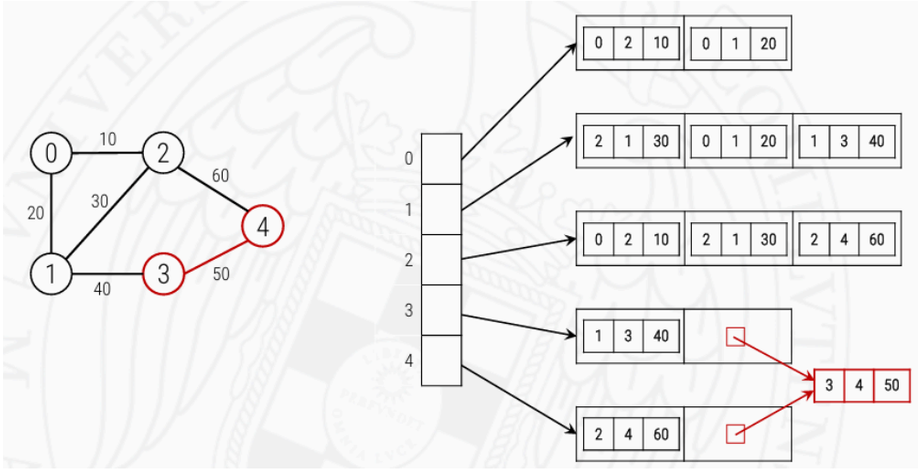
Grafos Valorados

Son grafos cuyas aristas tienen asociado un valor (peso, coste).



En las listas de adyacencia serán de un tipo `Arista`:

```
template <typename Valor>
class Arista {
public:
    Arista(int v, int w, Valor valor);
    int uno() const;
    int otro(int u) const;
    Valor valor() const;
    bool operator<(Arista<Valor> const& b) const;
    bool operator>(Arista<Valor> const& b) const;
};


 //int v = arista.uno(), w = arista.otro(v);
```



Implementación

```
template <typename Valor>
class GrafoValorado {
public:
    GrafoValorado(int V);
    void ponArista(Arista<Valor> arista);
    int V() const;
    int A() const;
    AdysVal<Valor> const& ady(int v) const;
    std::vector<Arista<Valor>> aristas() const;
};

void ponArista(Arista<Valor> arista) {
    int v = arista.uno(), w = arista.otro(v);
    if (v < 0 || v >= _V || w < 0 || w >= _V)
    throw std::invalid_argument("Vertice inexistente");
    ++_A;
    _ady[v].push_back(arista);
    _ady[w].push_back(arista);
 }

std::vector<Arista<Valor>> aristas() const {
    std::vector<Arista<Valor>> ars;
    for (int v = 0; v < V(); ++v)
          for (auto arista : ady(v))
                if (v < arista.otro(v))
                   ars.push_back(arista);
    return ars;
 }
```

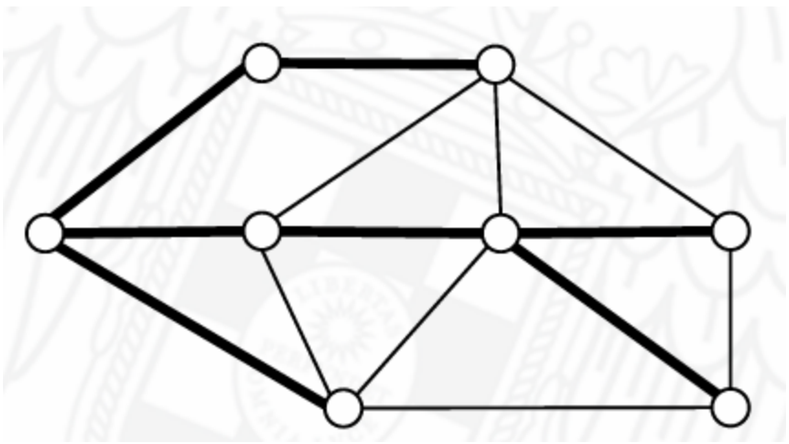
Recorrido en Profundidad

```
// visita los nodos alcanzables desde v respetando el umbral
void dfs(GrafoValorado<int> const& G, int v, int umbral) {
    visit[v] = true;
    for (auto a : G.ady(v)) {
        if (a.valor() < umbral) {
            int w = a.otro(v);
            if (!visit[w])
                dfs(G, w, umbral);
    }
}
```

Árboles de Recubrimiento de Coste Mínimo

Dado un grafo no dirigido G, un **arbol de recubrimiento** de G es un subgrafo T tal que:

- T es un árbol: es conexo y acíclico
- T es de recubrimiento: alcanza todos los vértices de G



Si T es un árbol de recubrimiento de un grafo G con vértices:

- T contiene exactamente V-1 aristas.
- Al eliminar cualquier arista de T deja de ser conexo.
- Añadir cualquier arista a T crea un ciclo.

El problema más común, es dado un grafo valorado no dirigido G, encontrar un árbol de recubrimiento de coste mínimo (ARM)

Aplicaciones

- Verificación facial en tiempo real.
- Búsqueda de redes de carreteras en imágenes de satélites o aéreas.
- Reducción del almacenamiento de datos en la secuenciación de aminoácidos de una proteína.

- Modelar la localidad de interacciones entre partículas en flujos de fluidos turbulentos.
- Algoritmos de aproximación para problemas NP-difíciles (por ejemplo, TSP, árbol de Steiner).
- Diseño de redes (comunicaciones, eléctricas, hidráulicas, informáticas, viales).

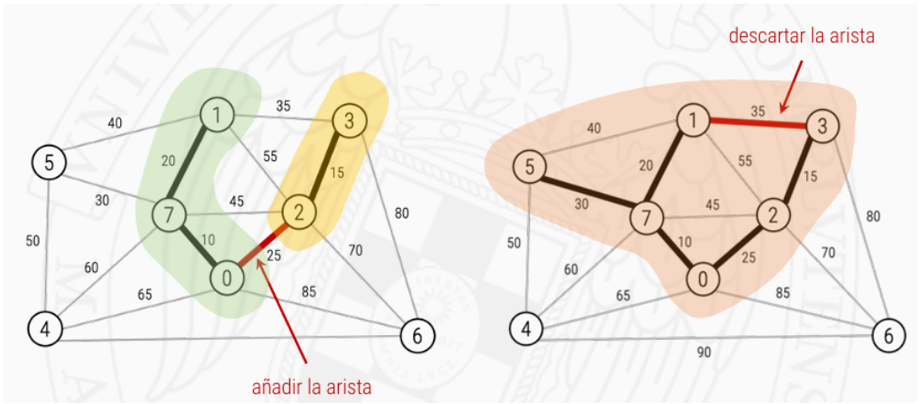
Algoritmo de Kruskal

Antes de desarrollar conviene aclarar varios conceptos:

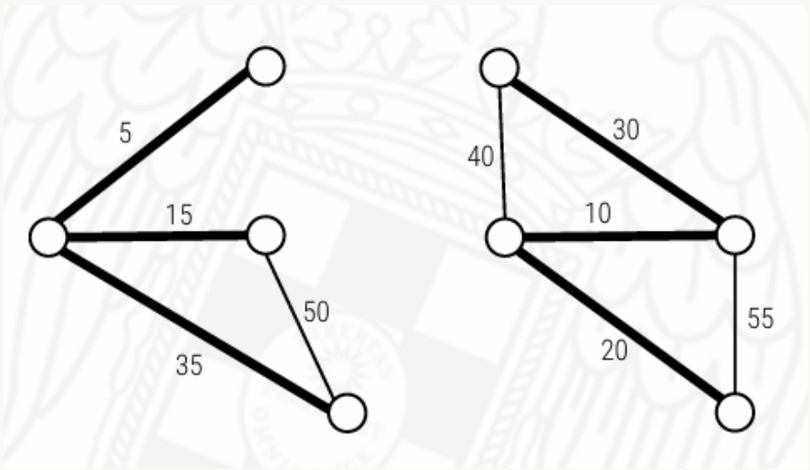
- Un **corte** de un grafo es una partición de sus vértices en dos conjuntos no vacíos.
- Una **arista cruza el corte** si tiene un extremo en cada conjunto.
- La **propiedad del corte**: dado un corte cualquiera, la arista de menor peso que lo cruza pertenece al ARM.

El algoritmo de Kruskal considera las aristas en orden creciente de coste, y cada arista se selecciona si no crea ciclos. Utiliza la propiedad del corte para conseguir la siguiente arista.

Para saber si hay ciclos o no al añadir una arista utilizamos los conjuntos disjuntos:



💡 Si el grafo no es conexo, se calcula un **bosque de recubrimiento mínimo**



💡 Si los costes de las aristas no son distintos, existe la posibilidad de que existan varios ARMs.

Implementación

```
template <typename Valor>
class ARM_Kruskal {
private:
   std::vector<Arista<Valor>> _ARM;
   Valor coste;
public:
   Valor costeARM() const {
      return coste;
   }

   std::vector<Arista<Valor>> const& ARM() const {
      return _ARM;
   }

   ARM_Kruskal(GrafoValorado<Valor> const& g) : coste(0) {
      PriorityQueue<Arista<Valor>> pq(g.aristas());
      ConjuntosDisjuntos cjtos(g.V());
      while (!pq.empty()) {
         auto a = pq.top(); pq.pop();
         int v = a.uno(), w = a.otro(v);
         if (!cjtos.unidos(v,w)) {
            cjtos.unir(v, w);
            _ARM.push_back(a); coste += a.valor();
            if (_ARM.size() == g.V() - 1) break;
         }
```

```
      }
    }
};
```

Coste

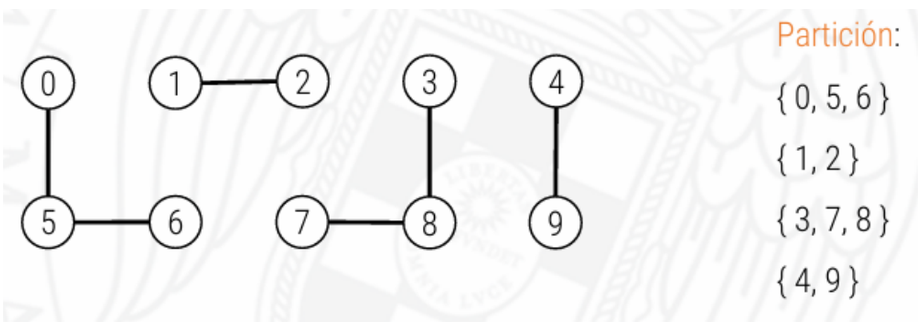
El algoritmo de Kruskal, aplicado a un grao con V vértices y A aristas, calcula el ARM en un tiempo en O(A logA) y con un espacio adicional en O(A).

Operación	Frecuencia	Coste por operación
construir cola prioridad	1	A
construir partición	1	V
`pop`	A	$\log A$
`unir`	$V - 1$	$\lg^* V$
`unidos`	A	$\lg^* V$

Conjuntos Disjuntos

Si quisiésemos representar un **relación de equivalencia** R:

- Reflexiva: a *R* a
- Simétrica: a *R* b ⇒ b *R* a
- Transitiva: a *R* b ^ b *R* c ⇒ b *R* a



TAD

El TAD de los conjuntos disjuntos cuenta con las siguientes operaciones:

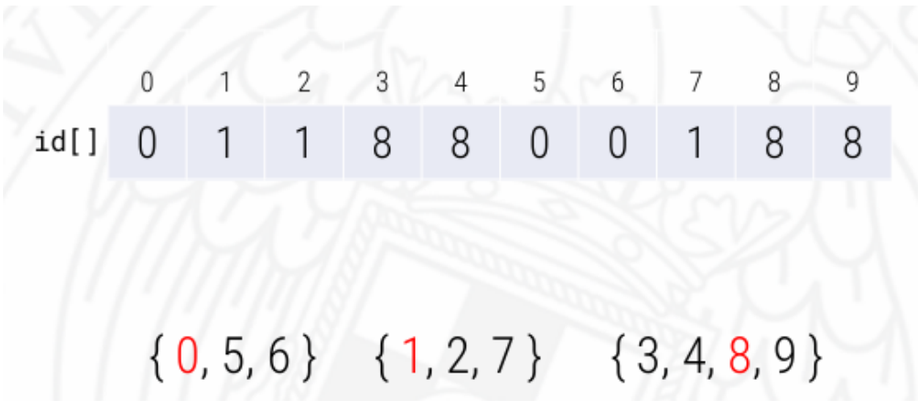
- crear una partición unitaria, `ConjuntosDisjuntos(int N)`
- unir dos conjuntos, `void unir(int a, int b)`
- buscar un elemento (devuelve el *representante* del conjunto), `int buscar(int a) const`
- consultar si dos elementos pertenecen al mismo conjunto, `bool unidos(int a, int b) const`
- consultar el cardinal de un conjunto, `int cardinal(int a) const`

- consultar el número de conjuntos, `int num_cjtos() const`

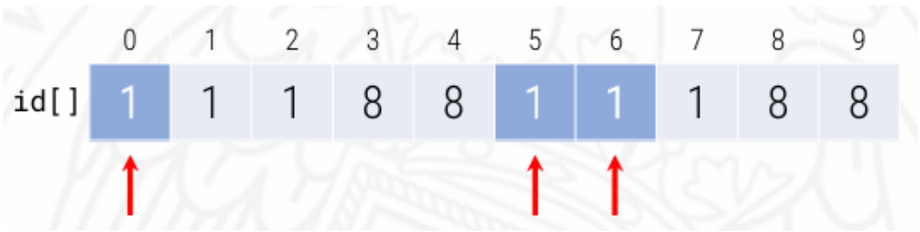
Hay dos implementaciones distintas

Búsqueda Rápida

Vector donde el índice de cada elemento es el representante de su conjunto.

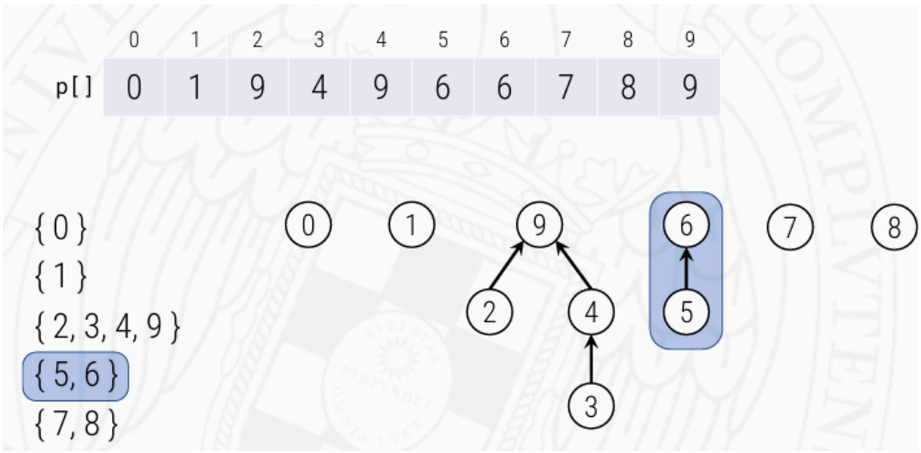


El problema es que la función de unir puede llegar a tener un coste O(N), porque si quisiesemos unir en nuestro ejemplo el 1 con el 6, tendríamos que cambiar todos los del conjunto del 6 (cuyo representante es 0) a 1:

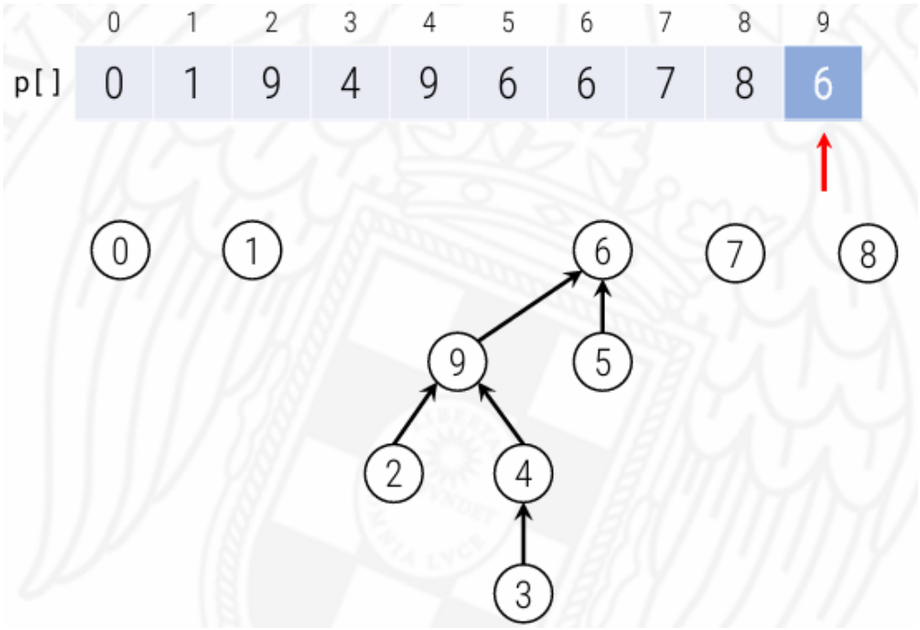


Unión Rápida

Cada conjunto se representa mediante un árbol, donde `p[i]` es el padre de `i`.



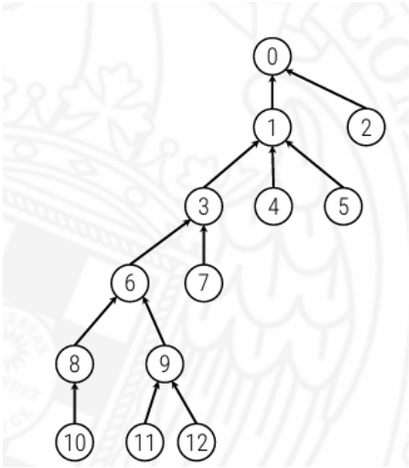
Entonces al unir solo cambiamos el padre del padre, es decir, si quisiesemos unir el 3 con el 5, tendriamos que poner al padre del 9 el 6:



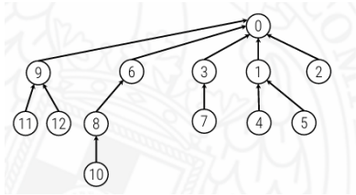
El problema, que puede ser que un árbol tenga tamaño N. Para solucionarlo mantendremos los tamaños de los árboles. Siempre pasará el arbol más pequeño a hijo del árbol mayor.

Al hacer la unión por tamaños, la profundidad será proporcional a la del elemento buscado, que estará acotada a O(logN)).

Después de buscar la raíz del árbol donde se encuentra x, cambiar su padre para que sea raiz:



→



Costes

Implementación	complejidad en el caso peor
búsqueda rápida	$N\ M$
unión rápida	$N\ M$
unión rápida por tamaños	$N + M \log N$
unión rápida con compresión de caminos	$N + M \log N$
unión rápida por tamaños y con compresión de caminos	$N + M \lg^* N$

M llamadas a `unir` y `buscar` sobre una partición de N elementos

N	$\lg^* N$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

Implementación

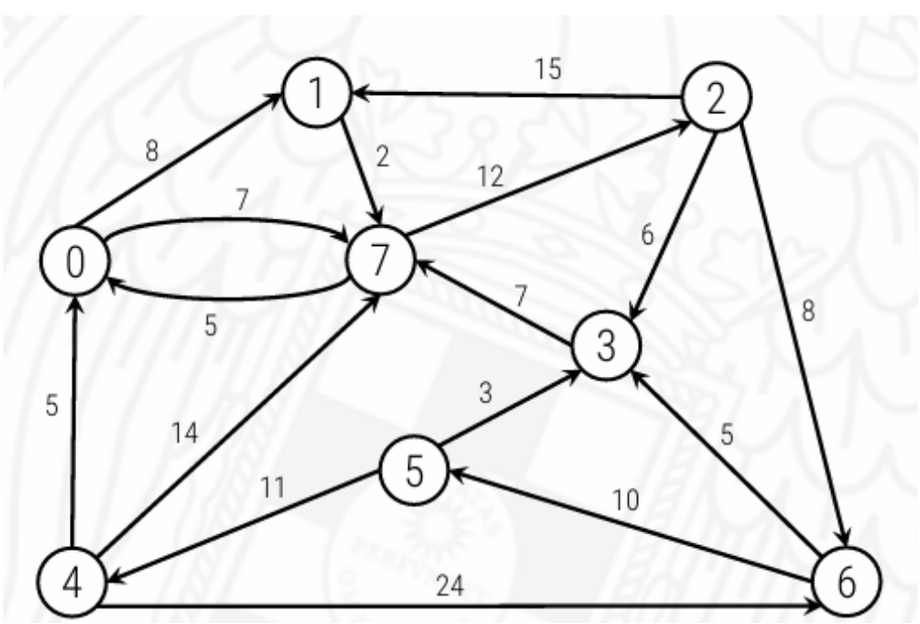
Se implementará la unión rápida por tamaños y con compresión de caminos.

```
class ConjuntosDisjuntos {
protected:
    int ncjtos; // número de conjuntos disjuntos
    mutable std::vector<int> p; // enlace al padre
    std::vector<int> tam; // tamaño de los árboles
public:
    // partición unitaria de N elementos
    ConjuntosDisjuntos(int N) : ncjtos(N), p(N), tam(N,1) {
        for (int i = 0; i < N; ++i)
            p[i] = i;
    }

    //Con M llamadas a unir
    void unir(int a, int b) { //O(N + M lg*N)
        int i = buscar(a);
        int j = buscar(b);
```

```
        if (i == j) return;
        if (tam[i] >= tam[j]) { // i es la raíz del árbol más grande
            tam[i] += tam[j]; p[j] = i;
        } else {
            tam[j] += tam[i]; p[i] = j;
        }--ncjtos;
    }

    //Con M llamadas a buscar
    int buscar(int a) const { //O(N + M lg*N)
        if (p.at(a) == a) // es una raíz
            return a;
        else
            return p[a] = buscar(p[a]);
    }

    bool unidos(int a, int b) const { //O(log*N)
        return buscar(a) == buscar(b);
    }

    int cardinal(int a) const { //O(log*N)
        return tam[buscar(a)];
    }

    int num_cjtos() const { //O(1)
       return ncjtos;
    }
};
```

Digrafos Valorados

Grafos con aristas orientadas que tienen asociado un valor (peso, coste).



En su representación de listas de adyacentes, cada Arista es representado por una clase `AristaValorada` que tiene un origen, un destino y un valor.

TAD

```
class DigrafoValorado {
public:
    void ponArista(AristaDirigida<Valor> arista) {
        int v = arista.desde(), w = arista.hasta();
        if (v < 0 || v >= _V || w < 0 || w >= _V)
        throw std::invalid_argument("Vertice inexistente");
        ++_A;
        _ady[v].push_back(arista);
       }
   DigrafoValorado<Valor> inverso() const {
        DigrafoValorado<Valor> inv(_V);
        for (auto v = 0; v < _V; ++v) {
            for (auto a : _ady[v]) {
                inv.ponArista({a.hasta(), a.desde(), a.valor()});
            }
        }
        return inv;
    }
....
}
```

Recorrido en Profundidad

```
void dfs(DigrafoValorado<int> const& g, int v, int ancho) {
    visit[v] = true;
    for (auto a : g.ady(v)) {
        if (a.valor() > ancho) {
            int w = a.hasta();
            if (!visit[w])
                dfs(g, w, ancho);
        }
    }
 }
```

Camino Mínimo

Dado un digrafo valorado, encontrar el camino mínimo de s a t.

Variantes del problema:

- Origen único: desde un vértice s a todos los demás

- Destino único: de cualquier vértice a un vértice t. (mismo enfoque del anterior pero con un grafo inverso)
- De punto a punto: de un vértice s a otro t (mismo enfoque que primero pero se puede parar cuando se encuentre t
- Entre cualquier par de vértices. (algoritmo de floid)

Restricciones sobre los pesos

- Pesos no negativos
- Pesos euclídeos
- Pesos arbitrarios

Presencia de ciclos

- Con ciclos dirigidos
- Sin ciclos dirigidos
- Sin ciclos de coste negativo

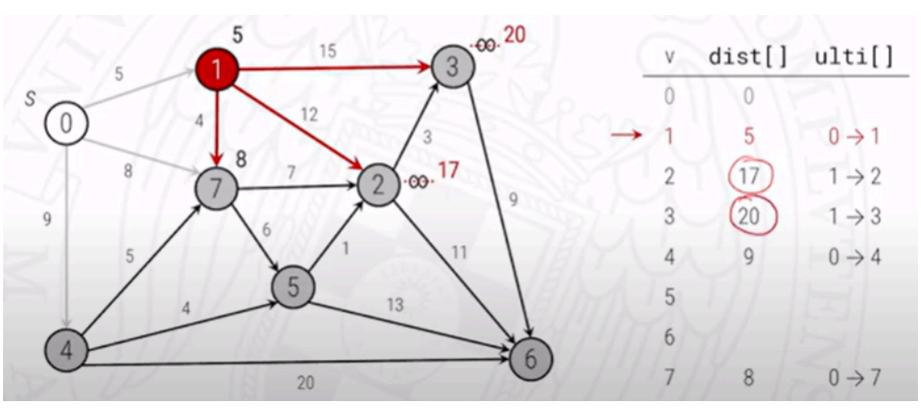
Los caminos mínimos forman un árbol de caminos mínimos.

Se pueden representar todos los caminos con dos vectores:

- `dist[v]` → es la longitud del camino más corto desde el origen a v
- `ulti[v]` → es la última arista del camino más corto desde el origen a v

Algoritmo de Dijkstra

- Considera los vértices en orden creciente de distancia desde el origen
- Añade el vértice al árbol y relaja todas las aristas que salen de él.
- El algoritmo se basa en estas condiciones de optimalidad
  - Sea `G` un digrafo valorado. `dist[]` contiene las distancias de los caminos más cortos desde s al resto de vértices si y solo si
    - `dist[s] = 0`
    - Para todo `v`, `dist[v]` es la longitud de algún camino de `s` a `v`
    - Para toda arista `v` → `w`, `dist[w]` ≤ `dist[v] + a.valor()`



El siguiente vértice a valorar siempre será el que menor distancia al origen tenga. Se utilizará una cola de prioridad variable.

Implementación

```
template <typename Valor>
 class Dijkstra {
 public:
    Dijkstra(DigrafoValorado<Valor> const& g, int orig) : origen(orig), dist(g.V(), IN
        dist[origen] = 0;
        pq.push(origen, 0);
        while (!pq.empty()) {
            int v = pq.top().elem; pq.pop();
            for (auto a : g.ady(v))
                relajar(a);
        }
 }

 bool hayCamino(int v) const { return dist[v] != INF; }

 Valor distancia(int v) const { return dist[v]; }

 Camino<Valor> camino(int v) const {
    Camino<Valor> cam;
    // recuperamos el camino retrocediendo
    AristaDirigida<Valor> a;
    for (a = ulti[v]; a.desde() != origen; a = ulti[a.desde()])
        cam.push_front(a);
    cam.push_front(a);
    return cam;
 }

private:
    const Valor INF = std::numeric_limits<Valor>::max();
    int origen;
    std::vector<Valor> dist;
    std::vector<AristaDirigida<Valor>> ulti;
    IndexPQ<Valor> pq;

    void relajar(AristaDirigida<Valor> a) {
        int v = a.desde(), w = a.hasta();
        if (dist[w] > dist[v] + a.valor()) {
            dist[w] = dist[v] + a.valor(); ulti[w] = a;
            pq.update(w, dist[w]);
        }
    }
```

```
};
```

- Dado un digrafo valorado con aristas de costes no negativos, el algoritmo de Dijkstra calcula un árbol de caminos mínimos desde un origen.
  - Cada arista `v` → `w` se relaja exactamente una vez (cuando se relaja v), haciendo que se cumpla `dist[w] ≤ dist[v] + a.valor()`
  - La desigualdad se mantiene hasta que termina el algoritmo, porque
    - `dist[w]` no puede aumentar
    - `dist[v]` no cambiará
- El algoritmo de Dijkstra, aplicado a un grafo con V vértices y A aristas, calcula caminos mínimos desde el origen al resto de vértices en un tiempo en O(A log V) y con un espacio adicional en O(V).

Operación	Frecuencia	Coste por operación
inicializar los vectores	1	V
construir cola prioridad	1	V
pop	V	log V
update	A	log V