Grafos

Técnicas avanzadas en programación

Prof. Rhadamés Carmona

Última revisión: 27/05/2019

Agenda

- Conceptos
- Representación
- DFS
- BFS
- Floyd-Warshall
- Flujo máximo en redes
- Grafos bipartitos

Conceptos

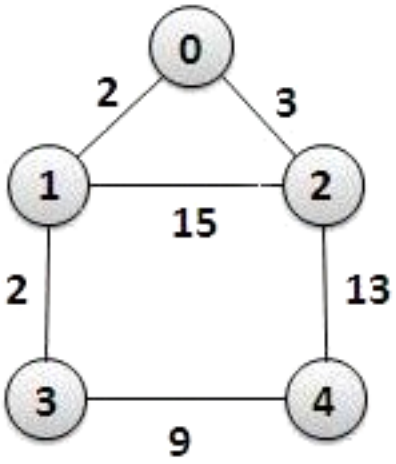
- Es un conjunto de elementos llamados vértices o nodos que está unidos por enlaces llamados aristas o arcos.
- Los arcos representan relaciones entre pares de vértices.
- G = (V, E), V=vértices, E = lados o *Edges*
- Grado($v \in V$) = arcos con extremo v
- Podemos tener grados dirigidos y no dirigidos
- Multigrafos: soporta más de una arista (a,b)

Conceptos

- Dos vértices son adyacentes si existe una arista que los une
- Dos aristas son adyacentes si comparten un vértice común
- Una arista puede tener un peso
- Vértices y aristas pueden etiquetarse

Representación

- **Matriz de adyacencia**: sea |V|=n. Se crea una matriz A[n][n], donde aij es el peso de la arista que une el vértice i y el j, o un valor “especial” (0, -1, INF, NaN, -INF) si la arista no existe.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Representación

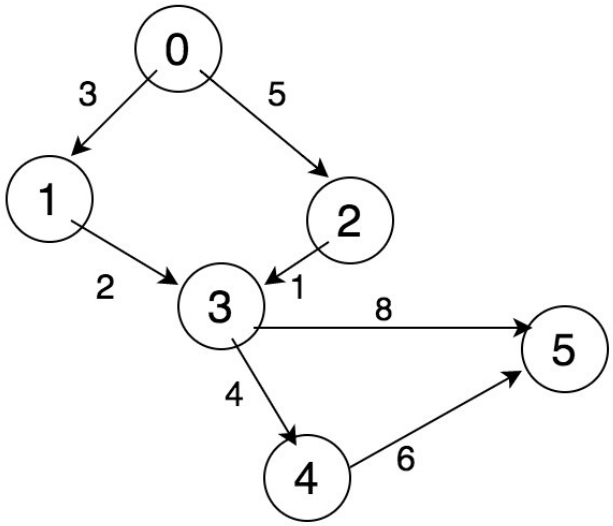
- **Matriz de adyacencia**:
  - Existencia de un arco es O(1)
  - Inserción/actualización de un arco es O(1)
  - Recorrer la adyacencia de un vértice es O(n=|V|)
  - Eliminar o insertar vértice es un problema con esta representación
  - Se puede almacenar la triangular superior en caso de grafo no dirigido

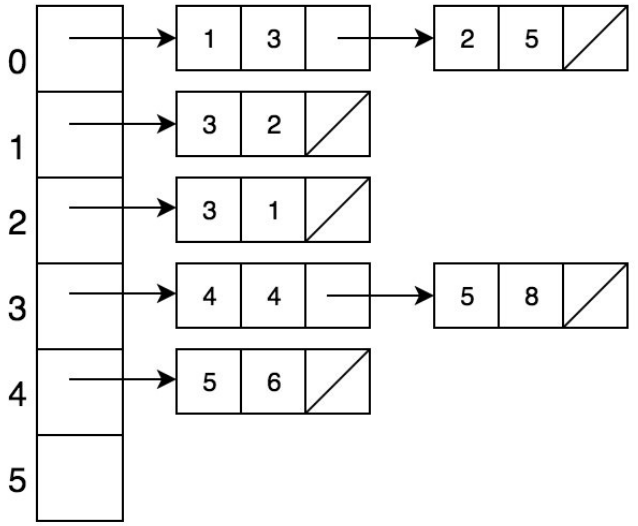
Representación

- **Lista de arcos**: se almacenan los arcos en una lista de pares {(1,2), (0,3), … }
- Insertar arco es O(1)
- Eliminar/buscar arco es O(m=|E|)
- Recorrer la adyacencia de un vértice es O(m=|V|)

Representación

- **Lista de adyacencia**: los vértices pueden almacenarse en un arreglo o en una lista; por cada vértice se tiene un arreglo o una lista de sus arcos. Cada arco tiene información del vértice adyacente (índice, apuntador o etiqueta), así como el peso del arco si aplica.





Representación

- **Lista de adyacencia**:
  - Inserción de arco es de O(1)
  - Búsquedas o eliminación de arco (a,b) son de O(|Ady(a)+Ady(b)|)
  - Recorrer a los vecinos de un vértice v es de O(|ady(v) |)

DFS

- Busqueda en profundidad o *Depth First Search*
- Recorre con backtracking los nodos, descubriendo otros nodos a través de la adyacencia. Los nodos adyacentes se recorren recursivamente, uno por uno. De allí el término depth first. No se considera el siguiente nodo hasta haber explorado todo lo explorable con el nodo actual.

DFS

- Problemas típicos que resuelve
  - Determinar si es grafo conexo
  - Determinar componentes conexas
  - Determinar la existencia de ciclos
  - Ordenamiento topológico
  - Biconexión de grafos
  - Componentes fuertemente conexas
  - Generación de laberintos

DFS

```
struct vertex
{
        edge *ady;
        int visited = -1;
        ...
};
```

```
typedef vector<vertex> graph;
int count;

void visit(graph  &g, int k) {
    g [k].visited = count++;          // marcar como visitado (0, 1, …)
    for (edge *p = g[k].ady; p!=NULL; p=p->next)
        if (g[p->v].visited == -1)
            visit(g,p->v);
}

void DFS(graph &g) {
    count = 0;
    for (int i=0; i<g.size(); i++)
        g[i].visited = -1;
    for (int i=0; i<g.size(); i++)
        if (g[i].visited == -1)  // undiscovered
            visit(g,i);
}
```

```
//También es posible definir graph:
class graph: public vector<vertex>
{
        ...
};
```

```
struct edge
{
        int v; // vert index
        float weight;
        edge *next;
};
```

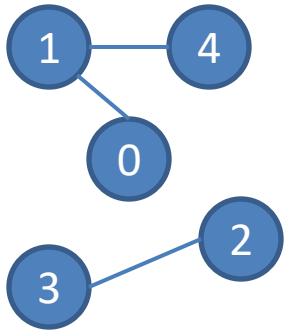
DFS

- Complejidad:
  - Utilizando la lista de adyacencia, se recorre el grafo visitando cada nodo 1 vez, y cada arco una vez: $O(|V| + |E|) \equiv O(n+m)$
  - Utilizando la matriz de adyacencia, $O(|V|^2) \equiv O(n^2)$, pues por cada nodo hay que visitar las n columnas en la matriz.

DFS

- **Contar componentes conexas**: basta contar cuantas llamadas a visit(i) se hacen desde la función DFS. Cada llamada a visit(i) significa que encontramos un nodo no alcanzable desde algún nodo j<i. Por lo tanto, es un pivote para descubrir una nueva componente conexa.

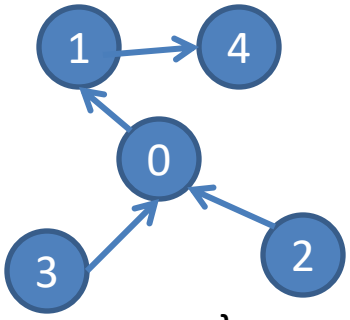
```
int DFS(graph &g) {
        count = 0;
        int components =0;
        for (int i=0; i<g.size(); i++)
                g[i].visited = -1;
        for (int i=0; i<g.size(); i++)
                if (g[i].visited == -1) {
                        components++;
                        visit(g,i);
                }
        return components;
}
```

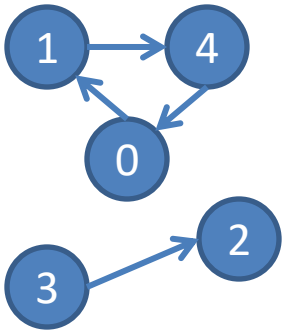


DFS

- **Existencia de ciclos en grafos dirigidos**: si encontramos un nodo dentro de “visit” que NO haya visitado todos sus hijos, encontramos un ciclo

```
void visit(graph &g, int k, bool &loop) {
        g[k].visited = 1;       // marcar como visitado
        for (edge *p = g[k].ady; p!=NULL && !loop; p=p->next)
                if (g[p->v].visited == -1)
                        visit(g, p->v);
                else if (g[p->v].visited == 1)
                        loop = true;
        g[k].visited = 2;  // visitó todos los hijos
}
```



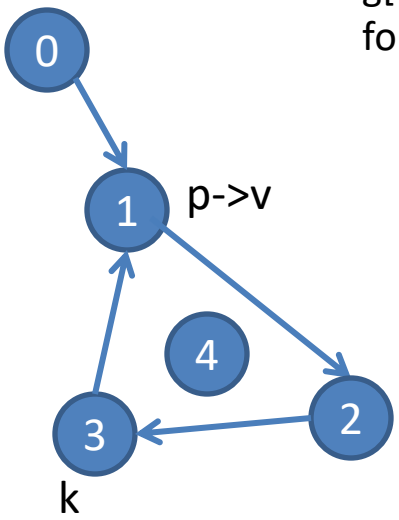


DFS

- **Existencia de ciclos en grafos dirigidos**: si queremos “obtener” el ciclo, podemos ir actualizando un “arreglo” o una “hash table” que nos indique por cada nodo, quién lo visitó: table[p->v]=k. Al encontrar el ciclo con un nodo p->v, ya sabemos que k trató de visitar a p->v. Buscamos en la hash quién visitó a k (k’=table[k]), luego quién visitó a k’ (k’’=table[k’]) y así hasta encontrar de nuevo a p->v. Esta reconstrucción del ciclo es O(*n*) usando un vector extra de *n* entradas (una entrada por vértice)

DFS

- **Existencia de ciclos en grafos dirigidos**: reconstruyendo el ciclo




```
bool visit(graph &g, int k, bool &loop) {
        g[k].visited = 1;          // marcar como visitado
        for (edge *p = g[k].ady; p!=NULL && !loop; p=p->next)
                if (g[p->v].visited == -1) {
                        visit(g, p->v);  table[p->v] = k;
                }
                else if (g[p->v].visited == 1) {
                        loop = true;  g[p->v].visited = IN_LOOP;
                         g[k].visited = IN_LOOP;
                        while(table[k] != p->v)  {
                                 g[table[k]].visited = IN_LOOP;
                                k=table[k];
                        }
                        return;
                }
         if (!loop) g[k].visited = 2;  // visitó todos los hijos
}
```

DFS

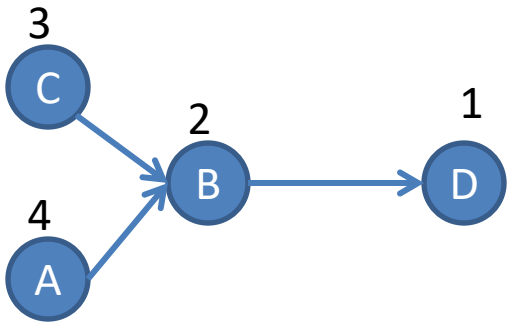
- **Ordenamiento topológico**: aplica a grafos acíclicos y dirigidos (DAGs).
- Consiste en ordenar los vértices “de izquierda a derecha” de manera tal que los arcos solo puedan ir de izquierda a derecha.
- Desde otro punto de vista, podemos re-etiquetar los vértices de manera tal que en cada arco (a,b) se tiene que a<b.

DFS

- **Ordenamiento topológico**: la utilidad es la realización ordenada de una secuencia de tareas, respetando las dependencias entre tareas (job scheduling)
- El ordenamiento topológico de un DAG no necesariamente es único.
- Solución 1: partir de nodos que no tengan arcos hacia él (sin dependencias), y enumerar los vértices en post-orden. Resulta el ordenamiento topológico inverso.

DFS

- Necesitamos saber qué vértices son fuentes. Debemos hacer un DFS por cada vértice "fuente" (no le llegan arcos).
- Note que el grafo debe ser acíclico!.



Ordenamiento topológico: A C B D (orden de visita: 4 3 2 1)
Una "tarea" no se ejecuta antes de las que la "preceden"

DFS

```
std::stack<int> stackObj; int count;

int DFS(graph &g) {
        count = 0;
        for (int i=0; i<g.size(); i++)
                g[i].visited = -1;
        for (int i=0; i<g.size(); i++)
                if (g[i].esFuente)
                        visit(g,i);
        print(stackObj);  // orden topológico = FILO
}

void visit(graph &gint k) {
        for (edge *p = g[k].ady; p!=NULL; p=p->next)
                if (g[p->v].visited == -1)
                        visit(g,p->v);
        stackObj.push_back(k);
         g[k].visited  = count++;
}
```

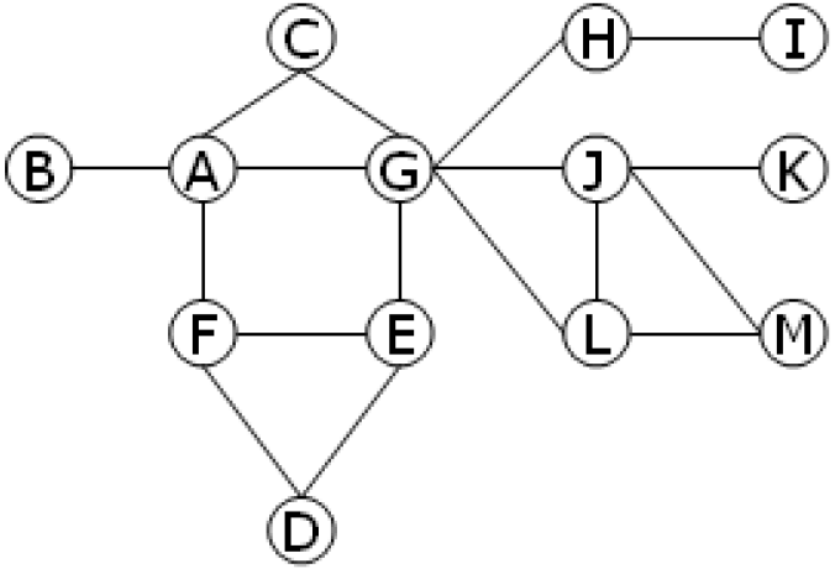
```
struct vertex
{
        edge *ady;
         int visited = -1;
         bool esFuente;
};
```

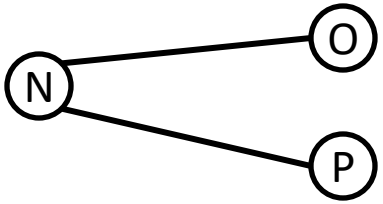
DFS

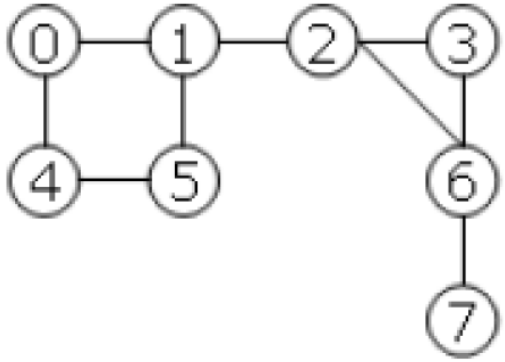
- **Puntos de articulación**: aplica a grafos conexos. Un vértices es un punto de articulación si al eliminarlo de un grafo conexo, el grafo queda disconexo (2 o más componentes conexas). El nodo debe tener al menos grado 2.
- **Solución**: para un vértice candidato k, chequear que algún hijo no tenga descendientes con un arco hacia un nodo etiquetado “antes de k”. Si no existe tal arco, entonces k es un punto de Articulación.

DFS

- **Puntos de articulación**: ejemplo





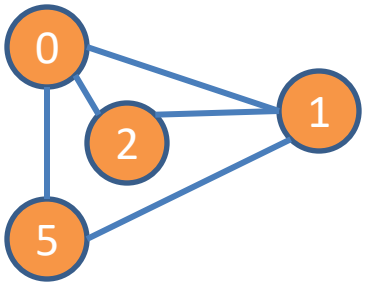


Puntos de articulación: A G H J N 1 2 6

https://www.eecs.wsu.edu/~holder/courses/CptS223/spr08/slides/graphapps.pdf

DFS

Se puede evitar pre-calcular el grado del nodo, y hacer el chequeo fuera del ciclo una vez sabido el grado?
Falta chequear root node (aparte)



- **Puntos de articulación:**

```
int visit(graph &g, int k, vector<int> &puntos_art) {
        int min=count;
        g[k].visited = count++;
        for (edge *p = g[k].ady; p!=NULL; p=p->next)
                if (g[p->v].visited == -1) {  // no visitado
                        int m = visit(g,p->v, puntos_art);
                        if (m >= g[k].visited && g[k].degree>=2)
                                puntos_art.push_back(k);
                        if (m<min) min=m; // mínima marca visitada
                }
                else if (g[p->v].visited < min)    // visitando antes del mínimo?
                        min = g[p->v].visited; // actualizo la marca mínima
        return min;
}
```

https://stackoverrun.com/es/q/4310072

DFS

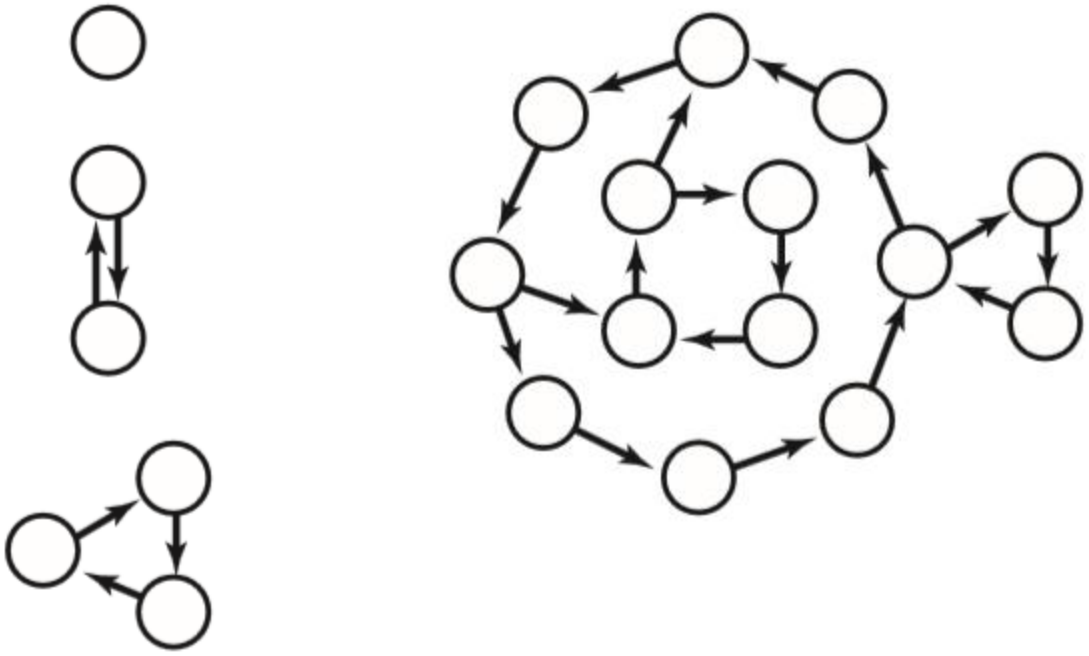
- **Grafo biconexo**: sucede cuando existe al menos dos caminos entre cada par de vértices.
- Un grafo conexo is biconexo si no tiene puntos de articulación.
- Para determinar que un grafo sea biconexo, hacemos un DFS, invocando a visitar solo una vez (todos los nodos deben ser alcanzables para que sea conexo) y no deben existir puntos de articulación (**puntos_art.size()==0**)

DFS

- **Componente fuertemente conexa** (CFC): una CFC en un grafo dirigido es un subgrafo en donde existe un camino entre cada par de vértices.
- Existe un algoritmo creado por Kosaraju (1978) y publicado luego por Sharir (1981) para extraer estas componentes en orden lineal O(|V|+|E|).

DFS

- **Componente fuertemente conexa** (CFC): cada par de vértices deben pertenecer a un ciclo dirigido.

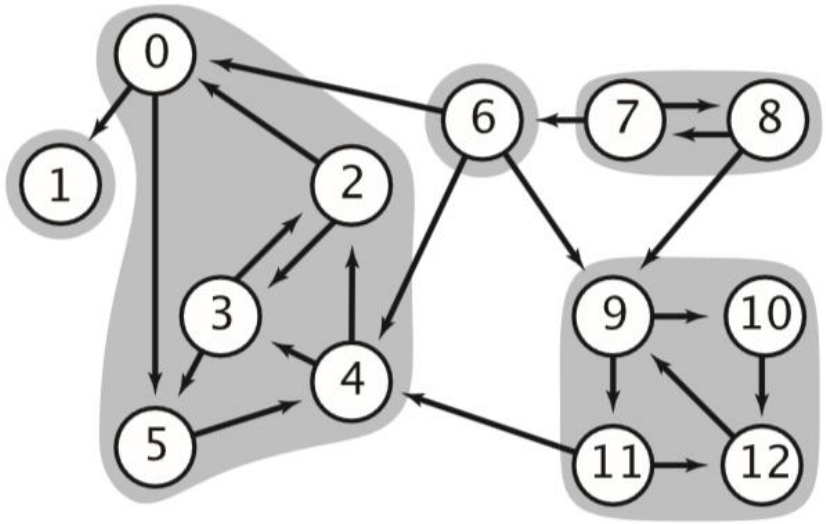


DFS

- **Componente fuertemente conexa** (CFC): entre las aplicaciones, los autores de un libro de texto deciden como agrupar los tópicos del libro; similarmente los desarrolladores de software puede decidir como particionar un software en módulos; incluso particionar un conjunto grande de páginas web que se relacionan, en tamaños más manejables.

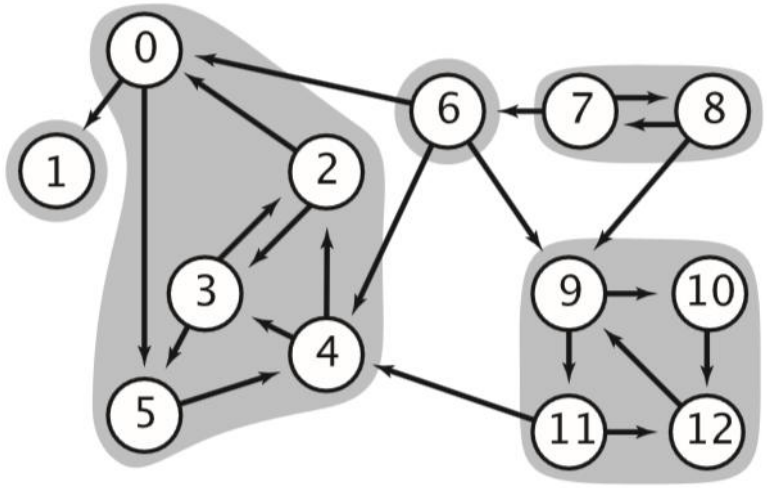
DFS

- **Componente fuertemente conexa** (CFC): cómo determinar las CFC's de un digrafo?
- Hay una relación de equivalencia entre los vértices de una CFC (reflexiva, simétrica y transitiva).



DFS

- **Componente fuertemente conexa** (CFC): podríamos hacer DFS por cada vértice en busca de ciclo. Luego todos los vértices de un ciclo pertenecen a la misma clase de equivalencia. Conjuntos disjuntos? Probar…

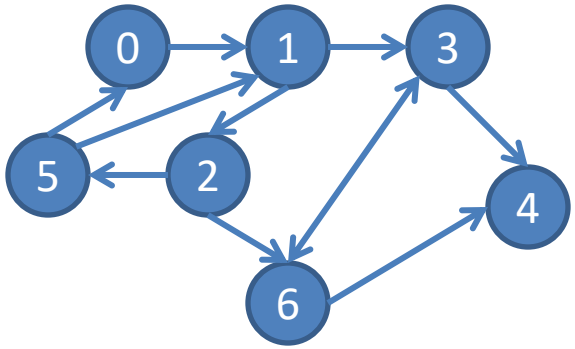


DFS

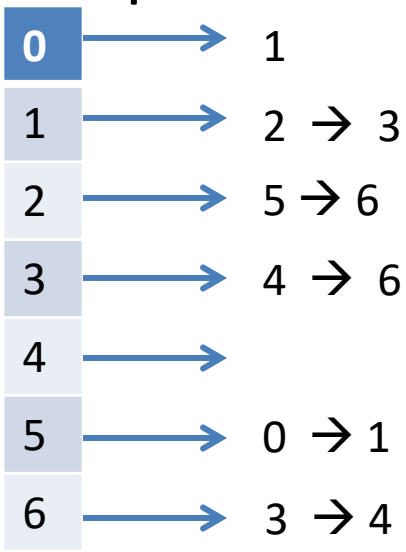
- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS convencional (visitando nodos alcanzables por cada vértice), introduciendo los vértices en post-orden en una pila.
  - Obtener el grafo “reverso” invirtiendo la dirección de todos los arcos. Desmarcar todos los vértices.
  - Hacer un DFS por cada vértice de la pila. Los nodos alcanzables en cada llamada a visitar forman una CFC (SCC = strongly connected comp.).

DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS convencional (visitando nodos alcanzables por cada vértice), introduciendo los vértices en post-orden en una pila.

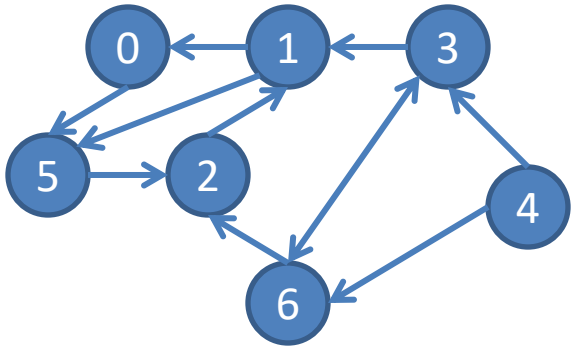


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

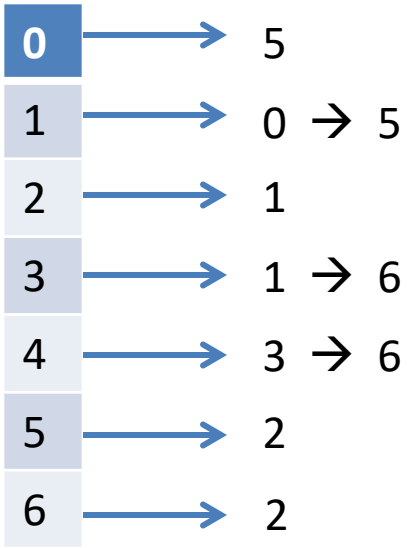


DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Obtener el grafo “reverso” invirtiendo la dirección de todos los arcos. Desmarcar todos los vértices.

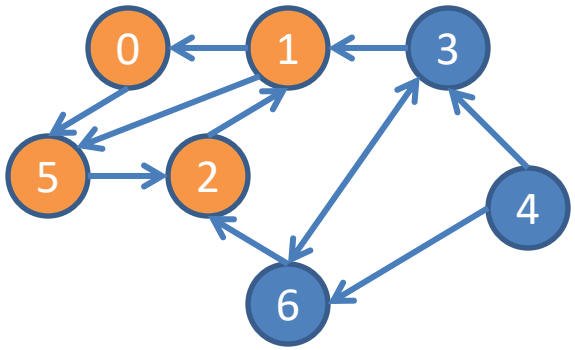


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

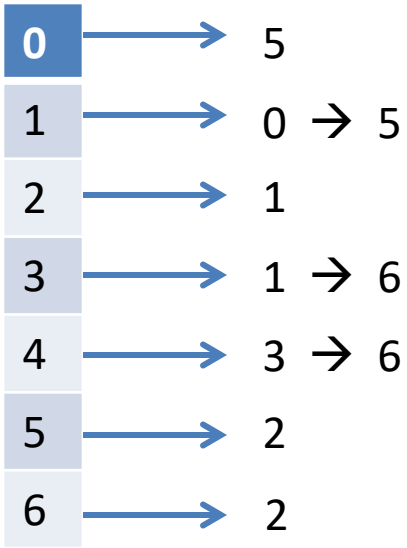


DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS por cada vértice de la pila. Los nodos alcanzables en cada llamada a visitar forman una CFC

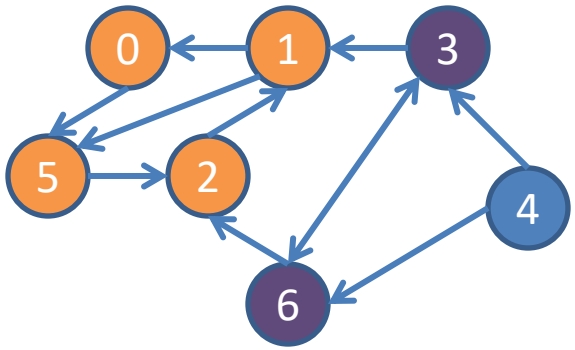


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)



DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS por cada vértice de la pila. Los nodos alcanzables en cada llamada a visitar forman una CFC

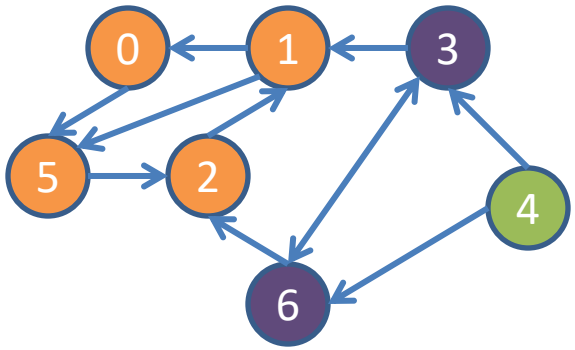


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	5
1	0 → 5
2	1
3	1 → 6
4	3 → 6
5	2
6	2

DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS por cada vértice de la pila. Los nodos alcanzables en cada llamada a visitar forman una CFC

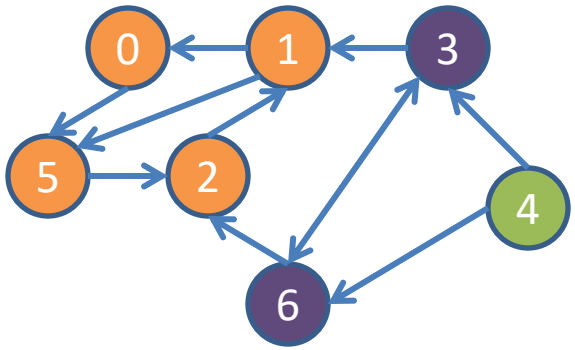


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

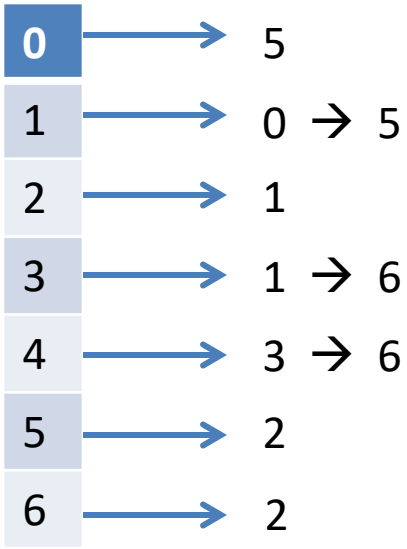
0	5
1	0 → 5
2	1
3	1 → 6
4	3 → 6
5	2
6	2

DFS

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)
  - Hacer un DFS por cada vértice de la pila. Los nodos alcanzables en cada llamada a visitar forman una CFC



P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)



DFS

```
struct vertex
{
        edge *ady;
        int visited = -1;
};
```

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)

```
std::stack<int> s; int count;

int DFS1(digraph &g) {
  count = 0;
  for (int i=0; i<g.size(); i++)
        g[i].visited = -1;
  for (int i=0; i<g.size(); i++)
        visit1(g,i);
}

void visit1(digraph &g, int k) {
   g[k].visited  = 1; edge *p;
  for (p = g[k].ady; p; p=p->next)
     if (g[p->v].visited == -1)
        visit1(g, p->v);
  s.push (k);
}
```

DFS

```
struct vertex
{
        edge *ady;
        int visited = -1;
};
```

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)

```
int DFS2(digraph &g) {
  digraph r = g.reverse();
  count = 0;
  for (int i=0; i<r.size(); i++)
        r[i].visited = -1;
  while (! s.empty()) {
    int x = s.top();
    if (r[x]->visited == -1) visit2(r, x);
    s.pop();
    count++;
  }
}
```

```
void visit2(digraph &g, int k) {
  g[k].visited  = count; edge *p;
  for (p = g[k].ady; p; p=p->next)
     if (g[p->v].visited == -1)
        visit2(g, p->v);
}
```

Cada nodo queda marcado con el número de CFC

DFS

- **Clausura transitiva:** La clausura transitiva de un dígrafo G es otro dígrafo con el mismo conjunto de vértices, pero con un lado de v a w en la clausura transitiva si y solo si w es accesible desde v en G.
- Se suele almacenar en una matriz, pues suele ser denso.
- Pensar en una solución para este problema. No existe algoritmo menor a n^2 que luego permita queries de ExitePath(a,b) en O*(1).

DFS

- **Ciclo Hamiltoniano**: Dado un DAG, diseñe un algoritmo de tiempo lineal para determinar si hay una ruta que visita cada vértice una vez.
  - **Respuesta**: Calcule una ordenación topológica y verifique si hay un arco entre cada par de vértices consecutivos en el orden topológico.

BFS

- **Breadth First Search, o búsqueda en anchura**: se utiliza una cola a la hora de descubrir nodos desde un vértice.
- El algoritmo empieza visitando e insertando el vértice en la cola.
- En cada paso se extrae la cabeza de la cola, y se visitan y encolan sus vértices adyacentes no visitados. Esto garantiza un recorrido nivel por nivel desde el vértice inicial.

BFS

Y si mejor visitamos al desencolar?

- **Breadth First Search, o búsqueda en anchura**:

```
void BFS(graph &g, int k) {
    for(int i = 0; i < g.size(); i++)
        g[i].visited = -1;
    list<int> q; // the queue
    // Mark as visited and enqueue it
    g[k].visited = 1; q.push_back(s);
    while(!q.empty()) {
        // Dequeue
        k = q.front();
        q.pop_front();
        for (edge *p = g[k].adj; p; p=p->next)
            if (g[p->v]->visited == -1) {
                g[p->v]->visited = 1;
                q.push_back(p->v);
            }
    }
}
```

BFS

- **Breadth First Search, algunas aplicaciones**:
  - Ruta más corta y árbol de expansión mínimo.
  - En redes sociales, buscar los amigos, amigos de los amigos, y sucesivamente hasta un nivel N.
  - Hacer broadcast en una red desde un nodo.
  - En flujo en redes, el algoritmo de Ford–Fulkerson.
  - Verificar si un grafo es bipartito.
  - Prim´s mínimo spanning tree.
  - Disjkstra.

BFS

- **Spanning tree**: o árbol de expansión mínimo para grafos no dirigidos y pesados (en los arcos), es un subgrafo conectado sin ciclos que incluye a todos los vértices.
- **Minimum spanning tree (MST)**: es el spanning tree de mínima suma de arcos. Podría no ser único. Aplicaciones: en redes de comunicaciones, hidráulicas, vías aéreas y terrestres, redes biológicas, etc. Asumiremos grafo conectado.
- Estudiaremos **Prim y Kruskal**

BFS

- **Prim's algorithm**: es un algoritmo de tipo voraz o greedy, que empezando desde un vértice cualquiera, agrega un arco a la vez al árbol. Selecciona el arco de mínimo peso que une un vértice del árbol con un vértice que no está en el árbol.

BFS

- **Prim's algorithm**: Cómo encontrar rápidamente el arco candidato?. Cada vez que se selecciona un arco a (y el vértice extremo v), se marca v como visitado, y se agregan los arcos (v,w) de ady(v) a la cola de prioridad (ordenada por peso). Algunos arcos podrían contener ambos vértices marcados. Estos serán ignorados por el algoritmo (actualización de la cola de prioridad lazy o perezosa).

```
struct vertex {
        edge *ady;
        bool visited = false;
};
```

BFS

```
struct edge {
        int  left, right;
        float weight;
        edge *next;
};
```

- **Prim's algorithm**:

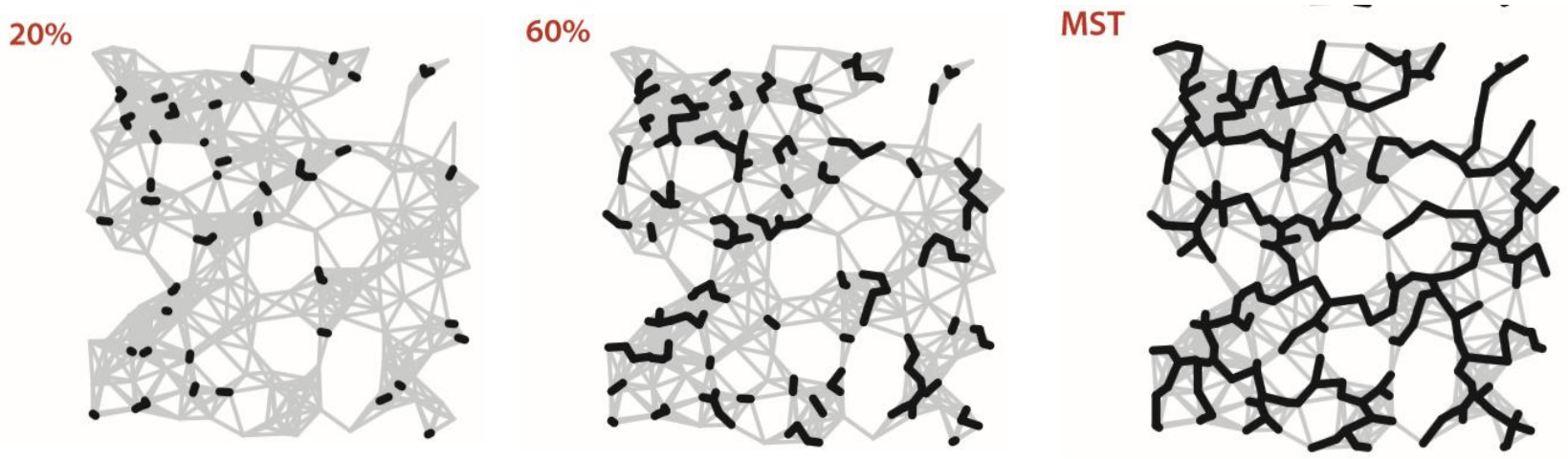
```
void LazyPrimMST(graph &g, list<edge *> &mst)   {       // O(m.logm)
        PriorityQueue<edge*> pq;  // ordenada asc. por edge::weight
        mst.clear();
        visit(g, 0, pq);
        while (!pq.empty())     {
           edge *e = pq.deleteMin();  // get lowest-weight
           if (g[e->left].visited && g[e->right].visited) continue;  //lazzy
           mst.push_back(e);
           if (!g[e->left].visited) visit(g, e->left, pq);       // add vertex to tree
           if (!g[e->right].visited) visit(g, e->right, pq);  // the same
        }
}
void visit(graph g, int v, PriorityQueue<edge *> pq)   {
   g[v].visited = true;
   for (edge *p = g[v].adj ; p != NULL; p=p->next)
      if ( ! g[p->right]->visited) pq.insert(e);
}
```

BFS

- **Kruskal algorithm**: en este caso, simplemente se ordenan todos los arcos ascendentemente por weight.
- Partiendo de un bosque de árboles de un nodo (todos los vértices), se agregan estos arcos uno por uno si no forman ciclo, hasta completar V arcos (V-1 vértices).
- Cada arco agregado “une” dos árboles del conjunto.

BFS

- **Kruskal algorithm**:



BFS

```
struct edge {
        int  left, right;
        float weight;
        edge *next;
};
```

- **Kruskal algorithm**:

```
Void KruskalMST(graph &g, queue<edge*> &mst)   {
  PriorityQueue<edge *> pq;  // heaps de mínimos
  insertAllEdges(pq, g);          // sorted by weght; O(m.logm)
  disjointSet <int>ds;            // conjuntos disjuntos
  while (!pq.empty() && mst.size() < g.size()-1)     {// O(m)
          edge *e = pq.deleteMin();                  // O(logm)
          if (ds.root(e->left) != NULL &&
              ds.root(e->left) == ds.root(e->right)) continue;
          ds.unionFind(e->left, e->right, true);   // Merge
          mst.push_back(e);
  }
}
```

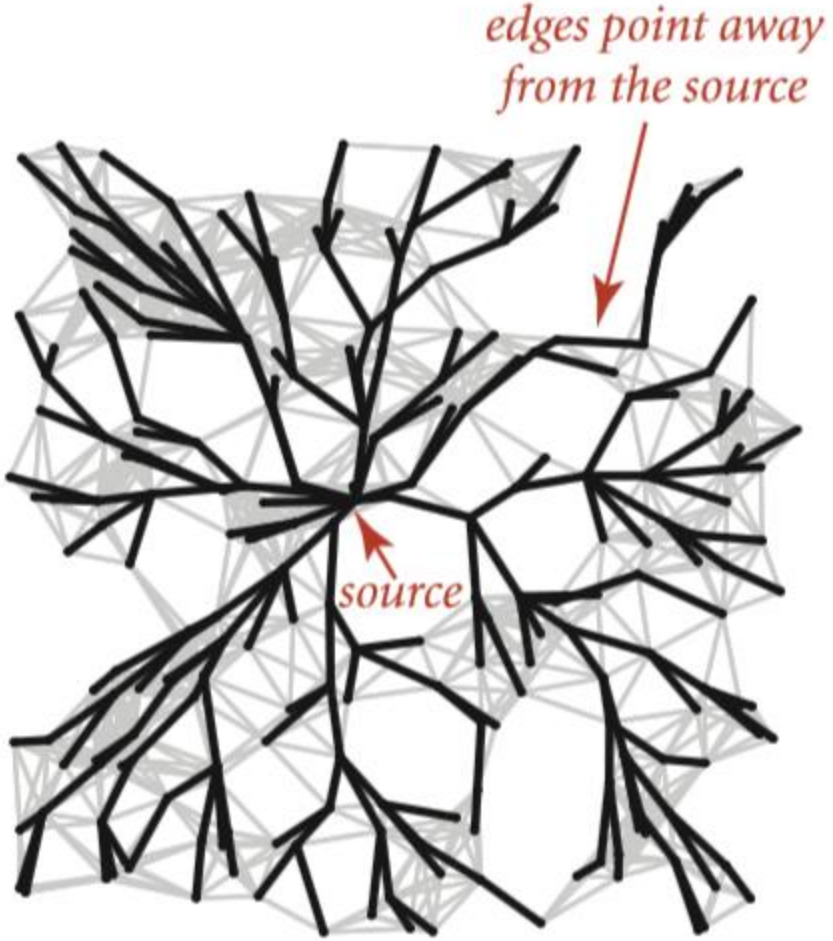
Recordemos que las operaciones de conjuntos disjuntos son O*(1)
Kruskal es entonces O(m.logm), con m = número de arcos

BFS

- **Shortest path:** encontrar el camino más corto entre par de vértices.
- Asumamos que es un digrafo, aunque también aplica para grafos.
- El peso del arco puede significar tiempo, longitud, costo, etc. En cualquier caso, es algo que se quiere minimizar. Asumiremos inicialmente pesos >= 0.

BFS

- **Shortest path:** se suele construir un shortest path tree; a partir de un vértice s, se genera un subgrafo que contiene a s, y específicamente un árbol rooted en s, donde cada camino desde s a cualquier vértice del árbol es el más corto en el grafo original. Luego se pueden hacer muchos queries desde s.

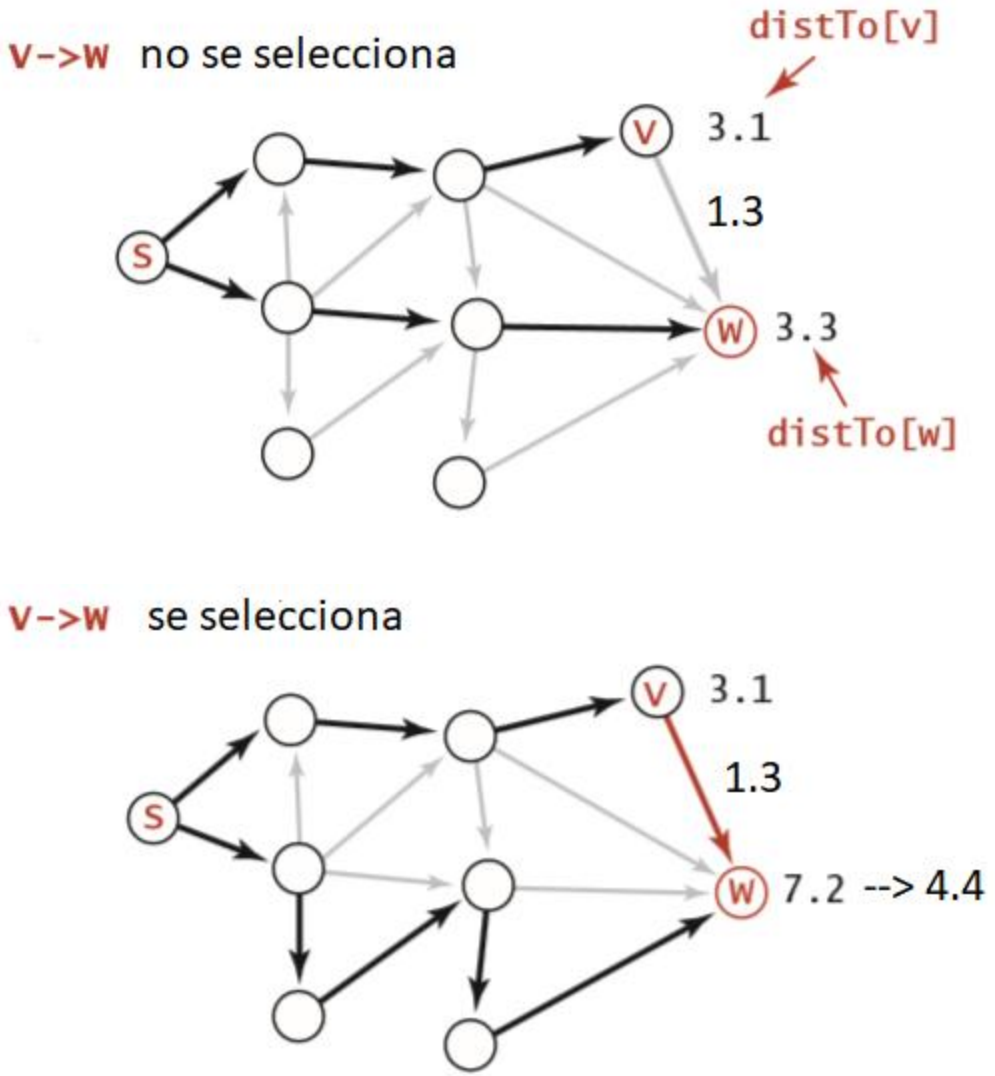


BFS

- **Shortest path**:
  - Sea edgeTo un arreglo de n enteros (índices a vértices), tal que edgeTo[v] contiene el índice al padre de v en el árbol.
  - Sea distTo un arreglo de n reales, tal que distTo[v] es la distancia mínima de s hasta v.
  - distTo[s] = 0.
  - Inicialmente, distTo[v] = ∞ (v $\neq$s), and edgeTo[v]= -1 para todo vértice v.
  - Hay 2 algoritmos clásicos: edge relaxation y vertex relaxation.

BFS

- **Shortest path (edge relaxation)**: en cada paso se considera un lado (v,w). Al procesar este lado simplemente se pregunta si es más corto pasar por el arco (v,w) o no?.



BFS

- **Shortest path (edge relaxation)**: un arco lo "relajamos":

```
vector<edge *> edgeTo;
vector<float> distTo;

void relax(edge *e) {
        int v = e->left;
        int w = e->right;
        if (distTo[w] > distTo[v] + e->weight)   {
                distTo[w] = distTo[v] + e->weight;
                edgeTo[w] = e;
        }
}
```

BFS

- **Shortest path (vertex relaxation)**: "relajamos" un vértice al relajar todos sus lados:

```
vector<edge *> edgeTo;
vector<float> distTo;

void relax(digraph &g, int v) {
   for (edge *p = g[v]->ady; p; p=p->next)   {
      int w = p->right;
      if (distTo[w] > distTo[v] + p->weight)  {
         distTo[w] = distTo[v] + p->weight;
         edgeTo[w] = v;
      }
   }
}
```

BFS

- **Shortest path (Dijkstra)**: similar al algoritmo Prim. La diferencia es que Prim selecciona el arco de mínimo peso que une un vértice del árbol con un vértice que no está en el árbol. Dijkstra en cambio agrega el vértice más cercano en distancia al “source” s.
- Empezamos encolando (en una cola de prioridad) al vértice s con distancia 0. Luego extraemos el primer vértice de la cola, “relajamos” el vértice, encolando los vértices adyacentes no visitados (o actualizando su peso si era visitado).

BFS

- **Shortest path (Dijkstra)**:

```
void Dijkstra(digrafo &g, int s) {
        for (int u=0; u<g.size(); u++) {
                distTo[u] = INFINITO ; edgeTo[u] = NULL; g[u].visited = false;
        }
        distTo[s] = 0; heap cola;
        cola.insert(make_pair<s, distTo[s]>);
        while (!cola.empty()) {
           int u = cola.min().index;   cola.desencolar();
           g[u].visited= true;
           for (edge *e = g[u].ady; e; e = e->next) /*if (!g[e->right].visited)*/ {
                   v = e->right;
                   if (distTo[v] > distTo [u] + e->weight) {
                           distTo [v] = distTo [u] + e->weight;
                           edgeTo [v] = u; // podría ser e, instead!
                           cola.insert(make_pair<v, distTo[v]>);
}       }  }       }
```

BFS

- **Shortest path (Dijkstra)**:

  - O(m.logn) usando cola de prioridad, pues cada arco (m) se inserta en una cola de prioridad de a lo sumo n vértices (logn)

  - Existe una implementación O(n.logn+m) usando heap de fibonacci

Floyd-Warshal

- **Floyd-Warshall**: el objetivo es tener una estructura que nos de el camino más corto entre cualquier par de nodos.
- Esto equivale a la clausura transitiva, pero almacenando como peso la distancia entre ambos vértices.
- Este algoritmo es n^3 si utilizamos la matriz de adyacencia. Se basa en la idea de **edge relaxation**.

Floyd-Warshal

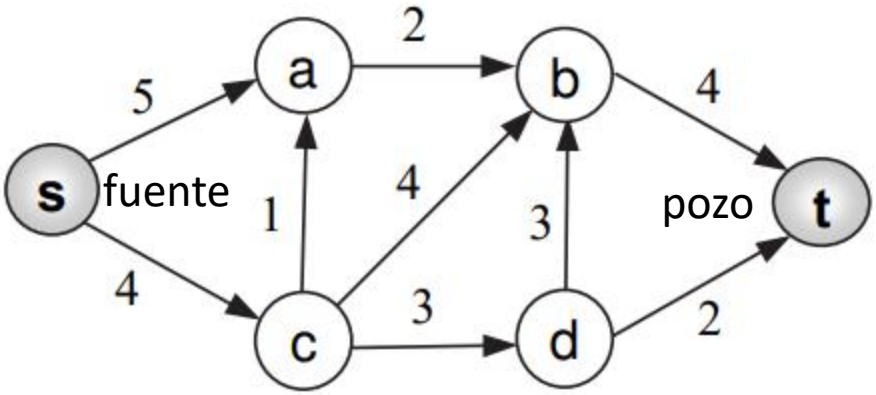
- **Floyd-Warshall**:

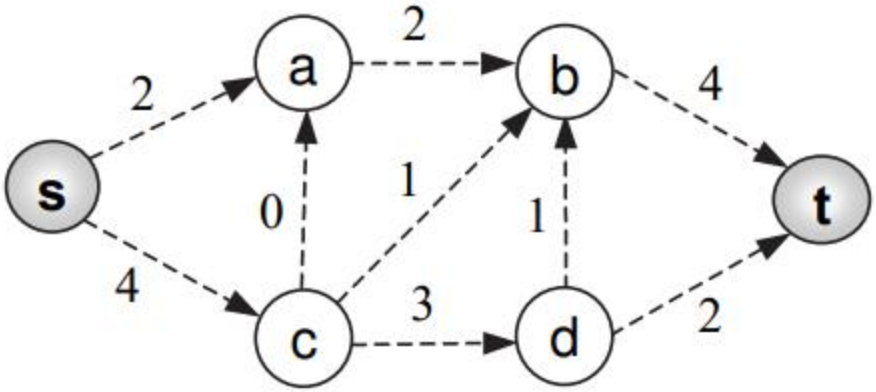
```
// matriz de adyacencia. Si el arco no existe, entonces vale infinito
// se asume además que ady[i][i] = 0 (diagonal cero)
// Devuelve una matriz con las distancias mínimas
void floydWarshall(int *ady[n], int *path[n]) {
          memcpy(path, ady, n*n*sizeof(int));
          for (int k = 0; k < n; k++)
                    for (int i = 0; i < n; i++)
                              for (int j = 0; j < n; j++){
                                        int d= path[i][k] + path[k][j];
                                        if (path[i][j] > d)
                                            path[i][j] = d;
                              }
}
```

Qué será mejor, n Dijkstras, o 1 Floyd-Warshall?

Flujo máximo en redes

- **Flujo máximo en redes:** El problema del flujo máximo consiste en calcular la cantidad máxima de flujo que puede ser transportada de un punto de partida u origen a uno de llegada o destino.





Flujo máximo en redes

- **Flujo máximo en redes:**
  - Hay dos vértices especiales: fuente y pozo
  - El peso del arco representa la capacidad del arco, el cual no puede excederse de ese valor
  - El valor final del flujo es el total de flujo que sale de la fuente, y equivalentemente la que llega al pozo
  - La capacidad puede pensarse como cantidad de agua en tubería, bandwidth, etc.

Flujo máximo en redes

- Solución naive: en cada iteración se verifica si existe un camino en la red residual (con capacidad > 0 por arco) desde la fuente hasta el pozo.
- De existir se aumenta el flujo en cada arco del camino en la red resultante según sea posible, mientras se reduce la capacidad de los arcos involucrados en la red residual.
- Se requiere llevar traza de la capacidad utilizada (red resultante) y de la capacidad remanente.
- El proceso termina cuando ya no exista otro camino.

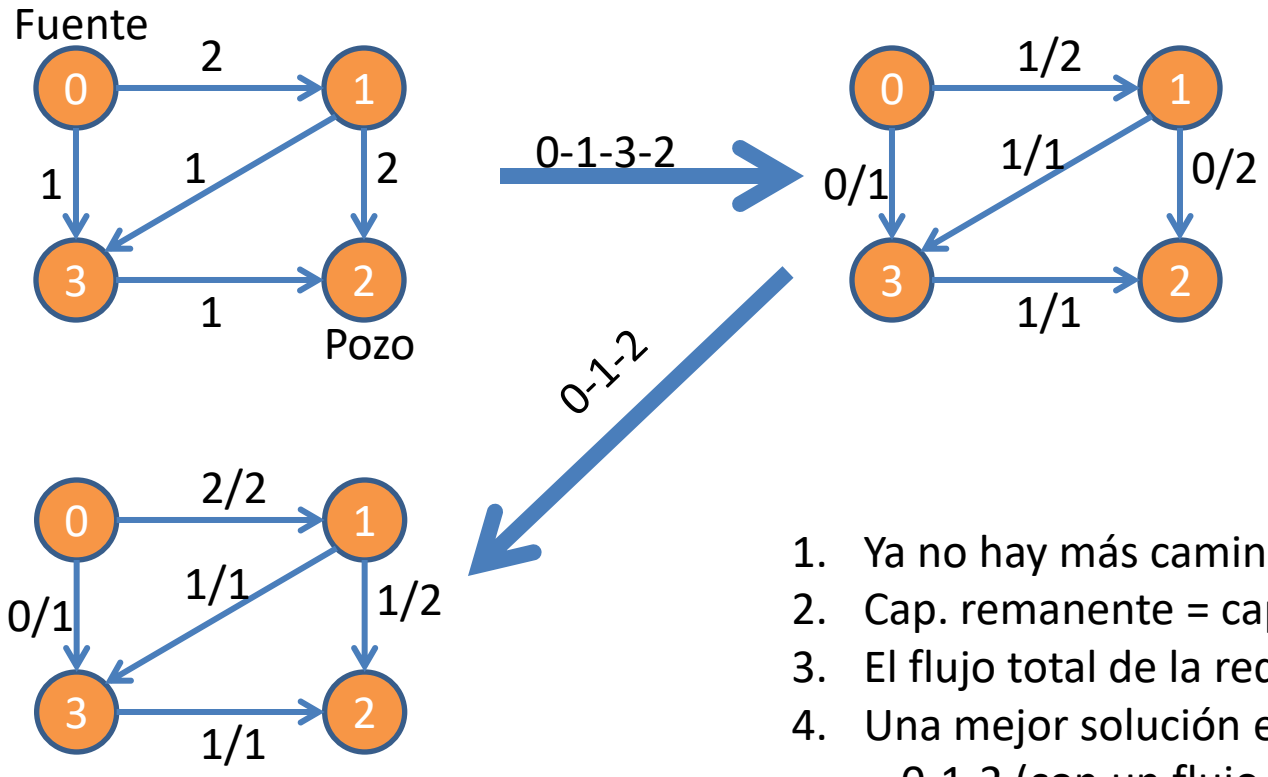
Flujo máximo en redes

- **Algoritmo naive:**

```
void naive(pair g[n][n], int s, int t) {  // NO ES OPTIMO!
        vector<pair> path;
        while (findPath(g, s, t, path))    {
                int min = g[path[0]][path[1]].capacity;
                for (int i=1; i<path.size()-1; i++)
                        if (min > g[path[i]][path[i+1]].capacity)
                                min = g[path[i]][path[i+1]].capacity;
                for (int i=0; i<path.size()-1; i++) {
                        int u = path[i], v = path[i+1];
                        g[u][v].capacity -= min; // quito capacidad
                        g[u][v].flow += min;  // agrego flujo usado
                }
        }
}
// para findPath, usar DFS BFS o A*
```

Flujo máximo en redes

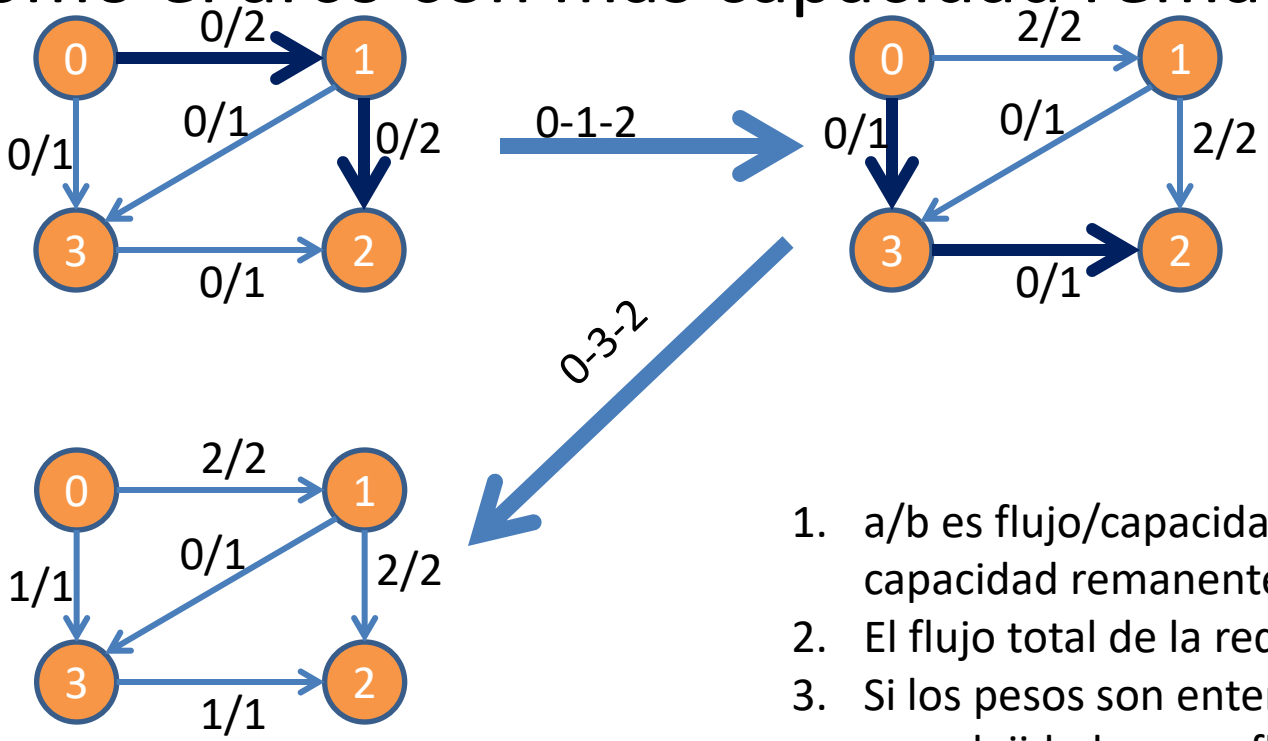
- **Algoritmo naive:** ejemplo



1. Ya no hay más caminos hacia el destino
2. Cap. remanente = cap. Inicial - flujo
3. El flujo total de la red es 2
4. Una mejor solución es elegir:
   0-1-2 (con un flujo de 2)
   0-3-1 (con un flujo de 1), sumando 3

Flujo máximo en redes

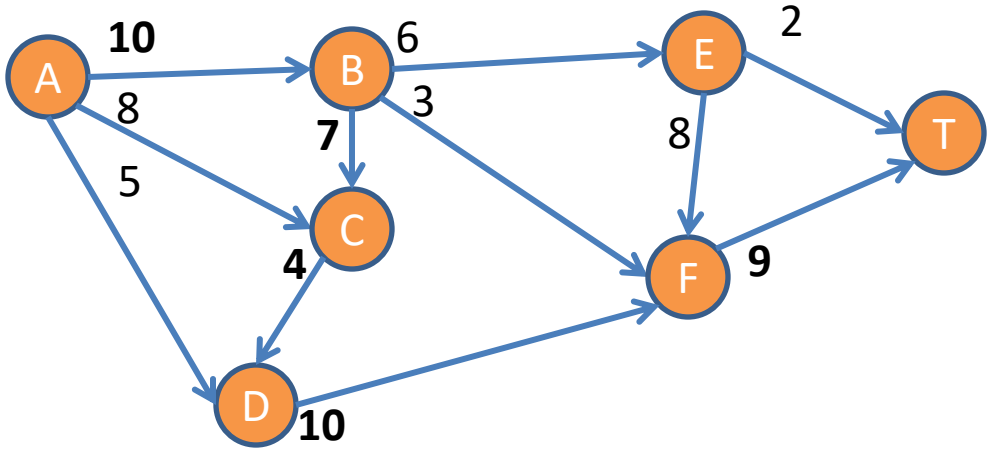
- **Algoritmo de flujo máximo:** en cada paso tomo el arco con más capacidad remanente



1. a/b es flujo/capacidad; la capacidad remanente es b-a
2. El flujo total de la red es 3
3. Si los pesos son enteros, la complejidad es maxflow*|E|

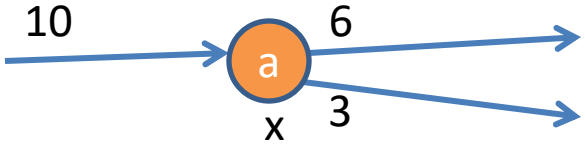
Flujo máximo en redes

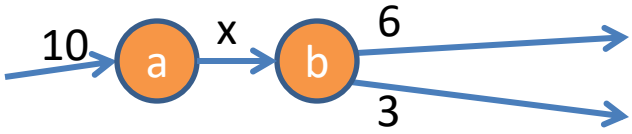
- **Algoritmo de flujo máximo**: en cada paso tomo el arco con más capacidad remanente



Flujo máximo en redes

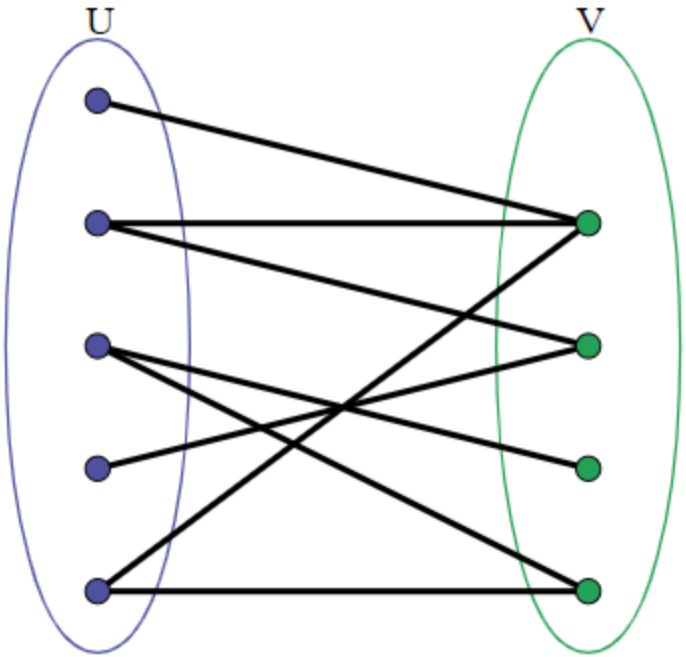
- **Algoritmo de flujo máximo**: si hay varias fuentes (o pozos), agregar una fuente (o pozo) ficticio con un enlace a cada fuente (o pozo) real. Cuál sería la capacidad hacia cada fuente (o pozo)?
- Si hay un “vértice con capacidad x”, lo reemplazamos por 2 vértices y un enlace con dicha capacidad





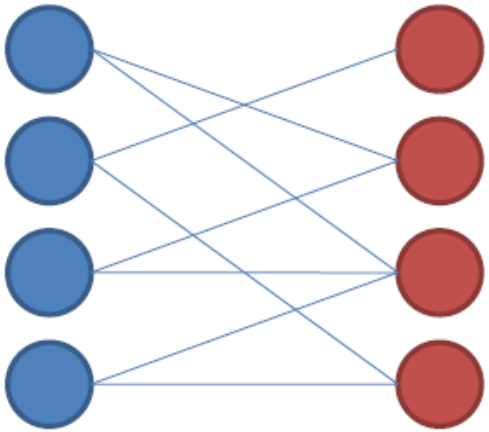
Grafos bipartitos

- Es un **grafo** cuyos vértices se pueden separar en dos conjuntos disjuntos U y V, donde no existen arcos que conecten 2 nodos de U, ni dos nodos de V, pero sí pueden existir arcos que conecten U con V.

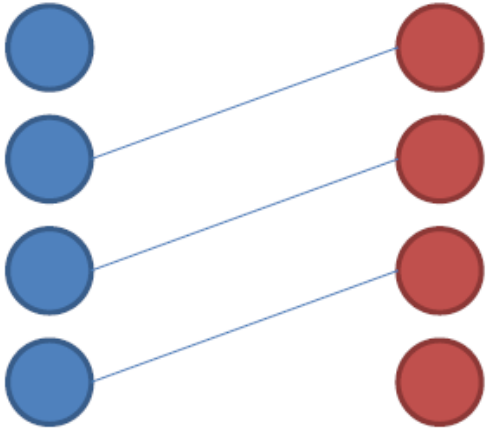


Grafos bipartitos

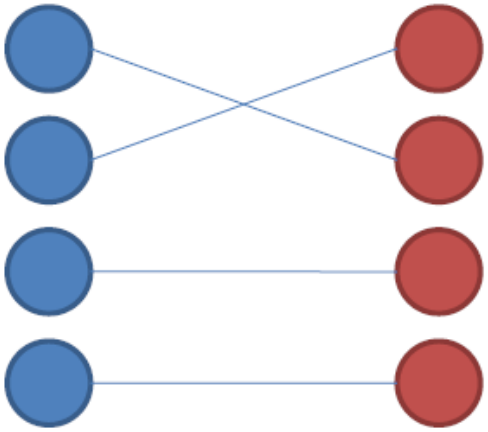
- Un “matching” consiste en seleccionar arcos de U hacia V de manera tal que cada vértice en U y en V tengan a lo sumo un arco adyacente.
- Maximum bipartite matching (MBM) significa que este matching tiene que ser máximo en número de arcos.



Grafo bipartito



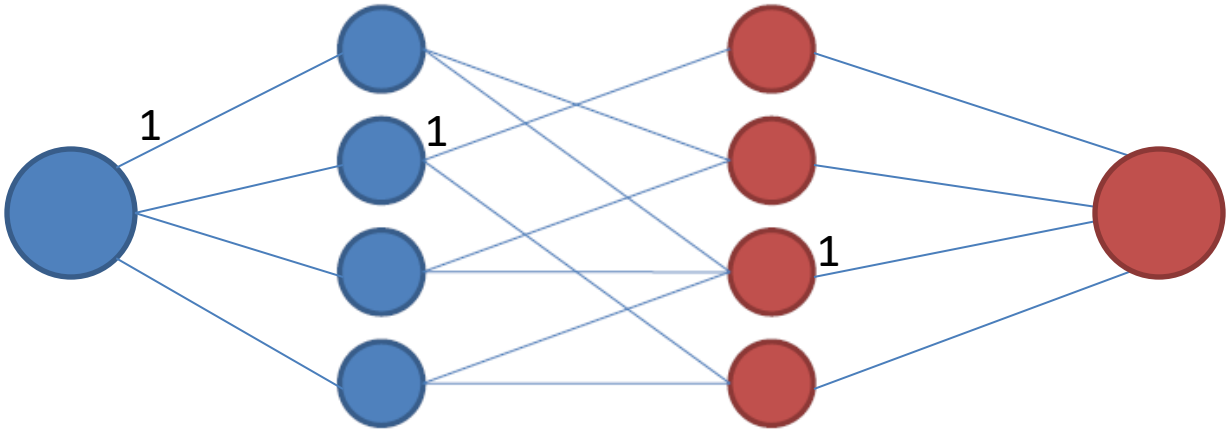
Un matching



Maximum bipartite matching

Grafos bipartitos

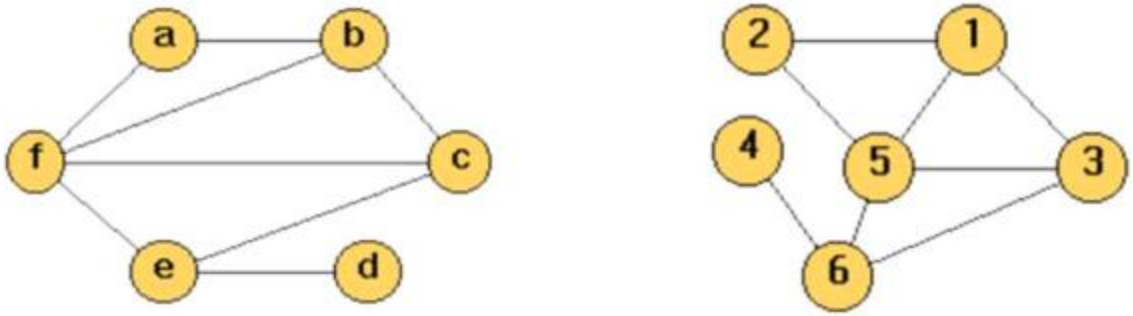
- Agregando una fuente y un pozo al grafo, convertimos el problema a “máximo flujo de red”



Se le coloca 1 al peso de cada arco

Otros problemas con grafos

- El grafo es planar? Es decir, se puede dibujar en 2D sin que los arcos se crucen? Ver Algoritmo de Hopcroft Tarjan de O(n+m)
- Isomorfismo de grafos: encontrar una biyección entre vértices de dos grafos, tal que ambos grafos sean equivalentes. Esto es igual a re-etiquetar los vértices de manera que ambos grafos sean iguales.



{(a→2), (b → 1), (c → 3), (d → 4), (e → 6), (f → 5)}

Otros problemas con grafos

- Min Cost Max Flow
- Min-Cut: corte mínimo en grafos
- Algoritmo Húngaro
- RMQ (Range Minimum Query), Lowest Common Ancestor (LCA), relación entre RMQ y LCA
- PERT (program evaluation and review technique) CPM (critical path analysis)
- En flujo en redes: Ford Furkenson, algoritmo de Edmonds-Karp, Push & Relabel