Grafos.

Jonathan Eidelman Manevich.
201310029010

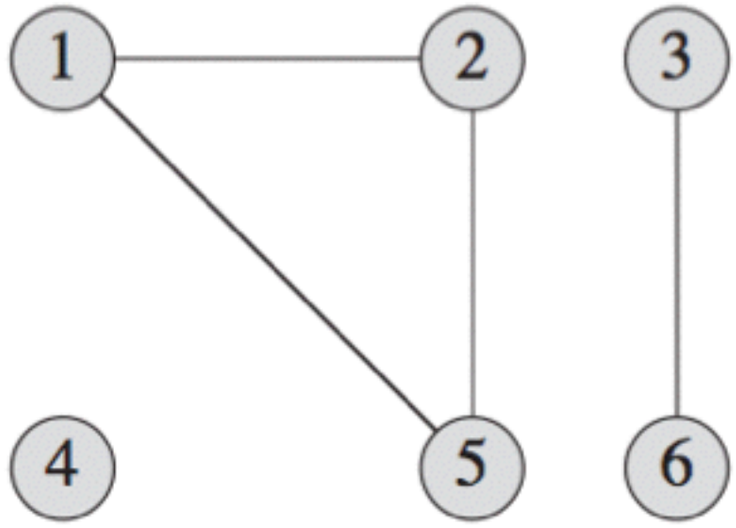
Introducción.

- Se define como G = (V,E) donde V es un conjunto finito de nodos (vértices) y E es un conjunto de parejas (Puede ser ordenadas o no ordenadas) (u,v) que pertenece a V.

No dirigidos.

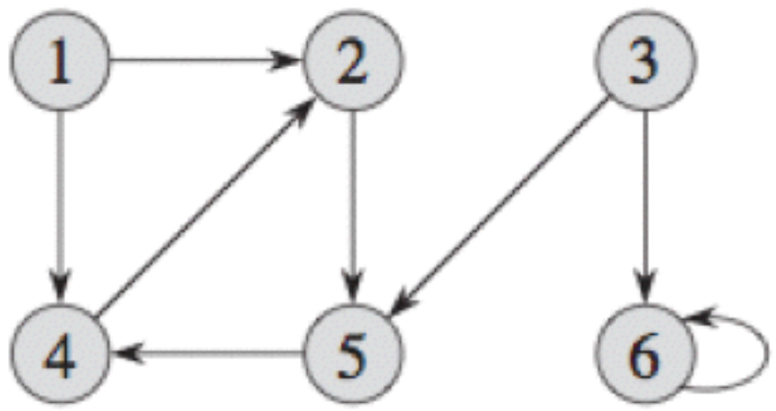
- En estos el conjunto E de parejas es no ordenado, pues es lo mismo el par (a,b) que el par (b,a).

En la siguiente figura $V = \{1, 2, 3, 4, 5, 6\}$ y
$E = \{(1, 2), (1, 5), (2, 5), (6, 3)\}$



Dirigidos.

- En estos el orden de las parejas si que importa, y si es posible tener una pareja del tipo (u,u).



Representación.

- Hay dos maneras de representar un grafo: Matriz de adyacencia o lista de adyacencia.

Matriz de adyacencia.

- Es una matriz indizada horizontal y verticalmente por los nodos existentes, generando así VxV “casillas”, en cada una de esas casillas podrá haber uno de dos valores: 1 ó 0.

- En caso de que el arco (arista) (a,b) exista en el grafo se debe llenar con un 1 la casilla correspondiente.

- En caso contrario se pone 0.

- Notar que en un grafo no dirigido tanto en la casilla [a][b] como la casilla [b][a] TIENE que haber el mismo contenido y la diagonal principal de la matriz TIENE que estar compuesta únicamente por 0. Esto implica que es una matriz simétrica.

Lista de adyacencia.

- Consiste en un arreglo de V vectores. para cada vértice u del grafo hay un vector con todos los vértices v tales que existe un par (u,v), es decir, todos los nodos adyacentes a u.

- Notar que en los grafos no dirigidos el par (u,v) tendrá dos apariciones, una en la lista de adyacentes de u, y otra en la de v.

Lista Vs. Matriz

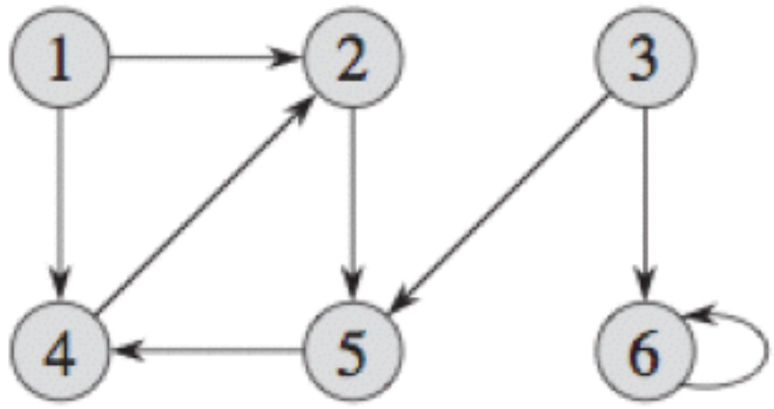
- Se recomienda usar la lista para grafos dispersos, cuando el número de aristas es pequeño (E >> |V| ^(2)).

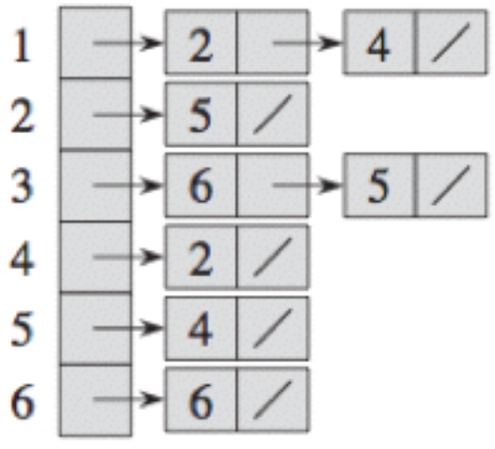
- La matriz se usa para grafos densos: E es cercano a |V|^(2).

- Verificar si un arco pertenece a un grafo es mucho más fácil en una matriz que en una lista de adyacencia.

- Una lista de adyacencia es recomendable cuando se quiere economizar en cuanto a memoria.

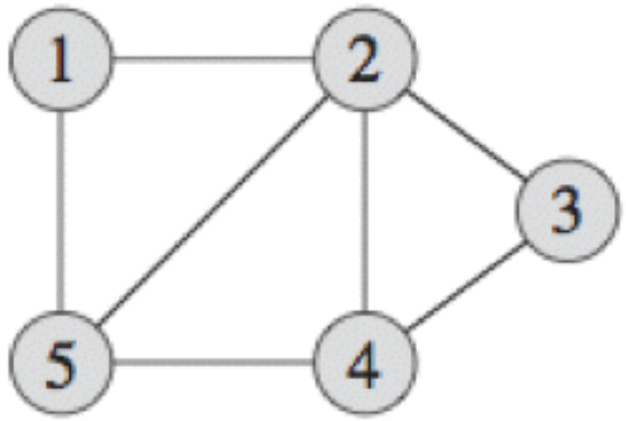
Visualización.

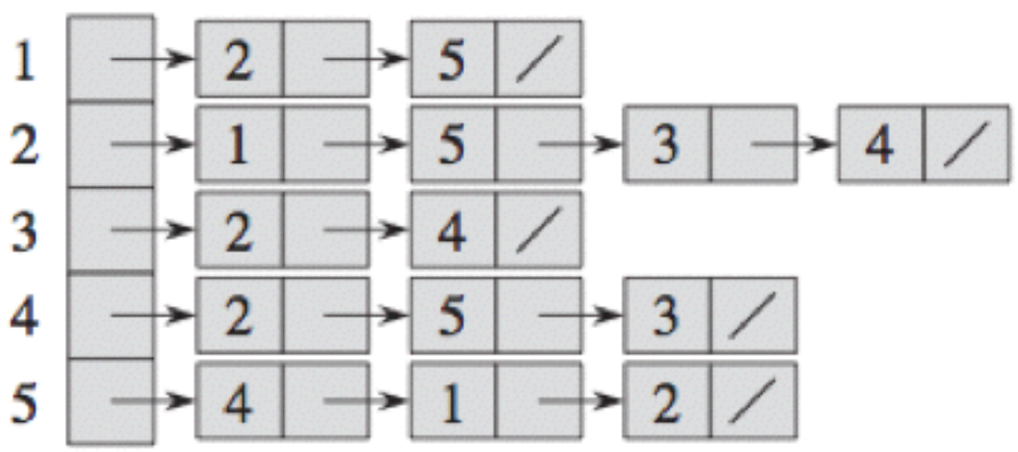




	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Visualización (2)





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Algoritmo (Ejemplo)

- Escribir un programa que genere las representaciones como matriz y lista de un grafo **no dirigido** G. La primera línea de entrada consiste de dos enteros n el número de nodos y m el número de aristas. Las siguientes m líneas contienen dos enteros u y v que indican que hay una arista que une el nodo u con v (o viceversa).

- (1<= n <= 10 , 1 <= m <= 100 , 1 <= u , 1 <= v)

Implementación -> Matriz.

```
1  using namespace std;
2  #include <iostream>
3
4  const int MAXN = 15;
5  int M [MAXN][MAXN];
6
7  int main(){
8      int n, m;
9      cin >> n >> m;
10
11     for (int i = 0; i < m; ++i){
12         int u, v; cin >> u >> v;
13         u--; v--;
14         M[u][v] = M[v][u] = 1;
15     }
16     return 0;
17 }
```

Implementación -> Lista.

```
1  using namespace std;
2  #include <iostream>
3  #include <vector>
4
5  const int MAXN = 15;
6  vector <int> g [MAXN];
7
8  int main(){
9      int n, m;
10     cin >> n >> m;
11
12     for (int i = 0; i < m; ++i){
13         int u, v; cin >> u >> v;
14         u--; v--;
15         g[u].push_back(v);
16         if (u != v) g[v].push_back(u);
17     }
18     return 0;
19 }
```

Warning!

- Como los “contenedores” están definidos de manera global, en un problema que hayan múltiples casos de prueba, entre cada uno de ellos se deben limpiar los “contenedores”.

Para la matriz.

```
1  using namespace std;
2  #include <iostream>
3
4  const int MAXN = 15;
5  int M[MAXN][MAXN];
6
7  int main(){
8      int n, m;
9      while (cin >> n >> m){
10         for (int i = 0; i <= n; ++i){
11             for (int j = 0; j <= n; ++j){
12                 M[i][j] = 0;
13             }
14         }
15         // Crear el grafo
16     }
17     return 0;
18 }
```

Para la lista.

```
1  using namespace std;
2  #include <iostream>
3  #include <vector>
4
5  const int MAXN = 15;
6  vector <int> g [MAXN];
7
8  int main(){
9      int n, m;
10     while (cin >> n >> m){
11         for (int i = 0; i <= n; ++i){
12             g[i].clear(); //Limpiar la lista del nodo i
13         }
14         // Crear el grafo
15     }
16     return 0;
17 }
```

Grafos con peso.

- En ocasiones se describen los arcos de los grafos con un peso, cuando esto ocurre en vez de llenar la matriz de adyacencia con 1 ó 0 se hace de la siguiente manera.

- M[u][v] = W(u,v) si (u,v) pertenece a E.

- de lo contrario, M[u][v] = +infinito.

Grafos con peso (2).

- La lista en estos casos se hace almacenando la **pareja** (v,W) en g[u], de existir el arco.

- en caso contrario, se almacenara en ese mismo lugar la **pareja** (v,+infinito).

Parejas.

- Es una estructura de C++ llamada pair.
- pair <type,type> someName.
- para guardar el grafo se debe hacer asi: vector<pair<int, int>> g[MAXN];
- las funciones first y second sirven para conocer los valores de la primera y segunda componente de la pareja respectivamente.

Recorridos.

- Existen dos recorridos.
- -> BFS stands for Breadth First Search.
- -> DFS stands for Depth First Search.

BFS.

- Sirve para recorrer o buscar elementos en un grafo.
- Se comienza en un nodo y se visitan todos los nodos vecinos a este.
- Para cada uno de esos vecinos (que no hayan sido explorados previamente) se aplica el mismo proceso.
- se continúa el proceso hasta que todo el grafo ya haya sido visitado.

Algoritmo.

```
1  vector <int> g[MAXN];    // La lista de adyacencia
2  int d[MAXN];             // Aristas usadas desde la fuente
3
4  void bfs(int s, int n){ // s = fuente, n = numero de nodos
5     // Marcar todos los nodos como no visitados
6     for (int i = 0; i < n; ++i) d[i] = -1;
7     queue <int> q;
8     q.push(s);                    // Agregar la fuente a la cola
9     d[s] = 0;                     // La distancia de la fuente es 0
10    while (q.size() > 0){
11       int cur = q.front(); q.pop();
12       for (int i = 0; i < g[cur].size(); ++i){
13          int next = g[cur][i];
14          if (d[next] == -1){        // Si todava no lo he visitado
15             d[next] = d[cur] + 1;  // La distancia que llevo + 1
16             q.push(next);          // Agregarlo a la cola
17          }
18       }
19    }
20 }
```

Aplicaciones.

- Hallar mínimo número de aristas para llegar desde la fuente hasta cualquier nodo.

- Hallar los nodos alcanzables desde la fuente.

- Si se guarda el nodo del que viene (padre) se puede tener el camino del mínimo número de aristas para llegar desde la fuente a cualquier nodo.

- Con algunas modificaciones sirve para ver si existe un ciclo en algún camino que sale desde la fuente.

Complejidad.

- con lista -> O(V+E)

- con matriz -> (V^(2))

DFS.

- Recorrer o buscar elementos en un grafo.
- Se comienza con un nodo y se marca como parcialmente visitado -> gris.
- se explora cada uno de los nodos vecinos de ese nodo.
- cuando termino de visitar todos los nodos vecinos, se marca el nodo como visitado -> negro.
- NO SE VISITA UN NODO HASTA HABER VISITADO TODOS SUS VECINOS.

Algoritmo.

```
1  vector <int> g[MAXN];        // La lista de adyacencia
2  int color[MAXN];             // El arreglo de visitados
3  enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3
4
5  void dfs(int u){
6     color[u] = GRAY;   // Marcar el nodo como semi-visitado
7     for (int i = 0; i < g[u].size(); ++i){
8        int v = g[u][i];
9        if (color[v] == WHITE) dfs(v);  // Visitar mis vecinos
10    }
11    color[u] = BLACK;  // Marcar el nodo como visitado
12 }
13
14 void call_dfs(int n){
15    // Marcar los nodos como no visitados
16    for (int u = 0; u < n; ++u) color[u] = WHITE;
17    // Llamar la funcion DFS con los nodos no visitados
18    for (int u = 0; u < n; ++u)
19       if (color[u] == WHITE) dfs(u);
20 }
```

Aplicaciones.

- Si se guarda el nodo padre, se puede hallar un camino desde la fuente hasta cada nodo.

- Ver si existe un ciclo en el grafo (En caso de que uno de los vecinos sea gris cuando lo voy a visitar).

- **Se puede hacer que retorne valores para verificar algunas características y condiciones del grafo.**

- **Si en vez de poner el color se pone el numero del grafo con el que se hizo la llamada inicial desde call_dfs, se puede hallar todos los nodos alcanzables desde ese nodo inicial.**

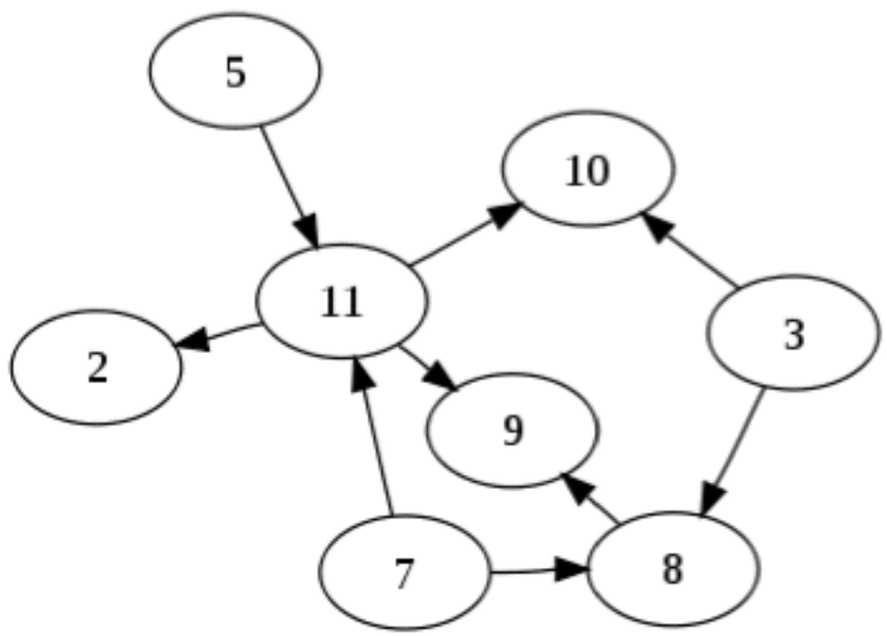
Complejidad.

- con lista -> O(V+E)

- con matriz -> (V^(2))

Sort Topológico.

- Un DAG (directed acyclic graph) es un grafo dirigido que no tiene ciclos.



Sort Topológico.

- Un ordenamiento topológico de un DAG es una organización lineal de sus nodos de tal forma que si (u,v) pertenece al conjunto de aristas, u aparece antes que v en el ordenamiento.

- Es una manera de poner todos los nodos en línea recta y todas las aristas irían de izquierda a derecha.

Algoritmo.

- Hacer un DFS con el grafo.
- Cuando marco un nodo como negro se inserta a un vector.
- Invertir el orden de los elementos del vector.
- Lo que esta ahora contenido en el vector es un ordenamiento topológico del grafo.

Algoritmo.

```
1  vector <int> g[MAXN];
2  bool seen[MAXN];
3  vector <int> topo_sort;
4
5  void dfs(int u){
6     seen[u] = true;
7     for (int i = 0; i < g[u].size(); ++i){
8        int v = g[u][i];
9        if (!seen[v]) dfs(v);
10    }
11    topo_sort.push_back(u); // Agregar el nodo al vector
12 }
13 int main(){
14    // Build graph: n = verices
15    topo_sort.clear();
16    for (int i = 0; i < n; ++i) seen[i] = false;
17    for (int i = 0; i < n; ++i) if (!seen[i]) dfs(i);
18    reverse(topo_sort.begin(), topo_sort.end());
19    return 0;
20 }
```

Funcionamiento.

- al insertar un nodo en la lista, previamente todos sus vecinos ya habían sido insertados.

- Si ya se procesaron todos los vecinos, ya se encuentra en la lista.

- Como todos los vecinos ya se habían ingresado, y luego se invierte el orden, en el ordenamiento los vecinos van a quedar posteriores a el nodo.

Complejidad.

- Sabemos que un DFS es V+E.
- Para reversar la lista es V.
- La complejidad sería el mayor entre esos dos ítems.
- La complejidad es O(V+E).

Componentes fuertemente conexas.

- Llamadas SCC por la sigla en inglés, es un subconjunto C de nodos que cumple que para cada pareja de nodos (u,v) existe un camino de u a v y viceversa y que C es lo mas grande posible.

Algoritmo.

- Crear el grafo G y un grafo auxiliar que será G con las aristas invertidas.

- Hacer DFS en G y generar su ordenamiento topológico.

- Hacer un DFS en G invertido, pero hacer las llamadas en el orden del ordenamiento topológico del paso anterior.

- Cada llamado a este DFS halla una componente fuertemente conexa.

Funcionamiento.

- Las SCC de G son las mismas que las de G invertido.

- si se comprimen los nodos de la SCC en un solo nodo se obtiene un DAG.

- Si se tienen dos SCC diferentes, C1 y C2, de manera que haya una arista de un nodo de C1 a un nodo de C2, entonces todos los nodos de C1 van a quedar después que los nodos de C2 en el ordenamiento topológico que se hace con el primer DFS.

Funcionamiento.

- Los nodos que quedan de primeros en el TS, son los nodos de una componente C a la cual no llega ninguna arista.

- En G invertido, de la componente C no salen aristas.

- Cuando se hace el llamado al segundo DFS se hace desde C y solo se descubren los elementos de C.

- Cuando se hace el llamado al segundo DFS desde otro nodo este puede no tener aristas salientes o sí tener aristas salientes a C pero como ya se descubrió todo de C solo se va a descubrir lo que hay en la componente de ese nodo.

Algoritmo de Dijkstra.

- Halla el camino mas corto desde una fuente s perteneciente a V a todos los nodos v pertenecientes a V cuando los pesos de las aristas son no negativos.

- Sea s el nodo fuente, d[v] la distancia desde s hasta v, p[v] el nodo predecesor a v en el camino mas corto de s a v. ————————> ver Algoritmo.

Algoritmo.

```
1  Hacer d[s] = 0 y d[v] = INF para todos los demas nodos
2  Hacer p[v] = -1 para todos los nodos
3  Agregar a la lista de nodos pendientes el nodo s con distancia 0
4  Mientras que haya nodos en la lista de pendientes
5     Extraer el nodo con la menor distancia (llamemoslo cur)
6     Recorrer cada vecino de cur (llamemoslo next)
7        Sea w_extra el peso de la arista de cur a next
8        Si d[next] > d[cur] + w_extra
9           d[next] = d[cur] + w_extra
10          p[next] = cur
11          Agregar next con d[next] a la lista de nodos pendientes
```

Implementación (1).

```
1  typedef pair <int, int> dist_node; // Tipo de dato para el heap
2  typedef pair <int, int> edge;      // Arista = pareja de enteros
3  const int MAXN = 100005;           // El maximo numero de nodos
4  const int INF = 1 << 30;           // Infinito = 2^30
5  vector <edge> g[MAXN];             // g[u] = (v, w)
6  int d[MAXN];    // d[u] La distancia mas corta de s a u
7  int p[MAXN];    // p[u] El predecesor de u en el camino mas corto
8
9  // La funcion recibe la fuente s y el numero total de nodos n
10 void dijkstra(int s, int n){
11    // Limpiar los valores de las variables
12    for (int i = 0; i <= n; ++i){
13       d[i] = INF;  p[i] = -1;
14    }
15    // Construir la cola de prioridades con la funcion mayor
16    priority_queue < dist_node, vector <dist_node>,
17                     greater<dist_node> > q;
18    d[s] = 0;
```

Implementación (2).

```
19      q.push(dist_node(0, s));
20      while (!q.empty()){
21          int dist = q.top().first;
22          int cur = q.top().second;
23          q.pop();
24          // No volver a procesar el nodo
25          if (dist > d[cur]) continue;
26          for (int i = 0; i < g[cur].size(); ++i){
27              int next = g[cur][i].first;
28              int w_extra = g[cur][i].second;
29              if (d[cur] + w_extra < d[next]){
30                  d[next] = d[cur] + w_extra;
31                  p[next] = cur;
32                  q.push(dist_node(d[next], next));
33              }
34          }
35      }
36  }
```

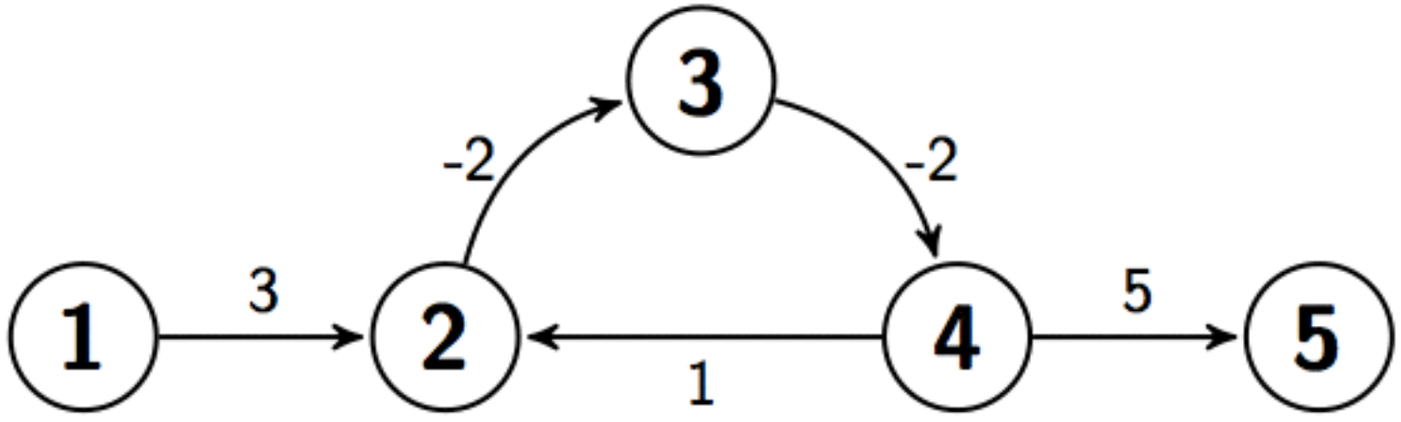
Obtener el camino mas corto.

```
1  vector <int> find_path (int t){
2      vector <int> path;
3      int cur = t;
4      while(cur != -1){
5          path.push_back(cur);
6          cur = p[cur];
7      }
8      reverse(path.begin(), path.end());
9      return path;
10 }
```

Algoritmo de Bellman-Ford.

- A veces es necesario encontrar el camino desde una fuente hasta todos los demás nodos en un grafo que tiene pesos negativo. Este algoritmo lo puede hacer.

¿Cuál es la distancia mas corta desde 1 hasta 5?



Algoritmo(1)

- Input: Un grafo G, una función de pesos W (posiblemente con elementos menores que 0), un nodo s perteneciente a V.

- Objetivo: para todo v perteneciente a V hallar el camino mas corto de s a v; ó decir que el grafo tiene un ciclo negativo y el problema no esta bien definido.

Idea principal

¿Cómo hacerlo?

Sea $L_{i,v}$ la longitud del camino más corto de s a v con máximo i aristas (se permiten ciclos).

$$L_{0,v} = \begin{cases} 0 & \text{si } v = s \\ +\infty & \text{si } v \neq s \end{cases}$$

Ahora $\forall v \in V$

$$L_{i,v} = \min \left\{ \begin{array}{c} L_{i-1,v} \\ \min\limits_{(u,v)\in E} \{L_{i-1,u} + w_{u,v}\} \end{array} \right\}$$

Cuando no hay pesos negativos.

- El camino mas corto tiene n - 1 aristas (Si es mayor que esto quiere decir que hay un nodo que se visitó mas de una vez y por lo tanto hay ciclos).

- Para hallar la solución se computa L(i,v) para i = 0,1, … , n - 1 y todo v perteneciente a V.

Algoritmo

```
1  Crear matriz L[MAXN][MAXN]  // L[i][u]
2  Hacer L[0][s] = 0 y L[0][u] = INF para u != s
3  Para i = 1 hasta n-1
4     // El camino ms corto con i-1 arstas
5     Para u = 0 hasta n-1
6        L[i][u] = L[i-1][u]
7
8     // Mirar todas las aristas
9     Para u = 0 hasta n-1
10       Para k = 0 hasta g[u].size() - 1
11          v = g[u][k].first  // El nodo
12          w = g[u][k].second // El peso
13          // Mirando la arista (u, v) de peso w
14          L[i][v] = min(L[i][v], L[i-1][u] + w)
```

Detección de ciclos de peso negativo.

- El grafo de entrada G tiene un ciclo de peso negativo sí y sólo si al hacer una iteración más del algoritmo se tiene que $L_{n,v} < L_{n-1,v}$

- Si lo anterior ocurre, quiere decir que usando n aristas se mejora la solución que se tenía al haberlo hecho con n - 1

- como el camino mas corto ya tiene n aristas, entonces el camino tiene n+1 nodos

- Esto implica que se repitió el uso de un nodo en el camino, lo cual implica un ciclo

- Esto se puede afirmar porque si el ciclo no fuese negativo, la solución no habría mejorado sino que se habría aumentado.

Algoritmo

- Ejecutar el algoritmo como si no hubieran ciclos de peso negativo.

- hacer una iteración más del algoritmo.

- Si en la nueva iteración se mejoró el resultado que se tenía, entonces se tiene la presencia de un ciclo de pesos negativos.

- De lo contrario no hay un ciclo de peso negativo.

Implementación

```
1  const int MAXN = 105;
2  const int INF = 1 << 30;
3  vector <pair <int, int> > g[MAXN];
4  int L[MAXN][MAXN];
5
6  bool bellman_ford(int s, int n){
7     for (int u = 0; u <= n; ++u) L[0][u] = INF;
8     L[0][s] = 0;
9     // Hallar la distancia mas corta con i aristas
10    for (int i = 1; i <= n - 1; ++i){
11       for (int u = 0; u < n; ++u) L[i][u] = L[i-1][u];
12
13       for (int u = 0; u < n; ++u){
14          for (int k = 0; k < g[u].size(); ++k){
15             int v = g[u][k].first;
16             int w = g[u][k].second;
17             L[i][v] = min(L[i][v], L[i-1][u] + w);
18          }
```

Implementación

```
19          }
20      }
21      // Verificar la existencia de ciclo de peso negativo
22      for (int u = 0; u < n; ++u) L[n][u] = L[n-1][u];
23
24      for (int u = 0; u < n; ++u){
25          for (int k = 0; k < g[u].size(); ++k){
26              int v = g[u][k].first;
27              int w = g[u][k].second;
28              // Si se mejora la solucion agregando una arista
29              if (L[n][v] > L[n-1][u] + w) return true;
30          }
31      }
32      // No hubo ciclo de peso negativo
33      // Distancia mas corta de s a u almacenada en L[n-1][u]
34      return false;
35  }
```

Implementación con optimización de espacio.

```
1  const int MAXN = 105;
2  const int INF = 1 << 30;
3  typedef pair <int, int> edge;
4  vector <edge> g[MAXN];
5  int d[MAXN];
6
7  bool bellman_ford(int s, int n){
8      for (int u = 0; u <= n; ++u) d[u] = INF;
9      d[s] = 0;
10
11     for (int i = 1; i <= n - 1; ++i){
12         for (int u = 0; u < n; ++u){
13             for (int k = 0; k < g[u].size(); ++k){
14                 int v = g[u][k].first;
15                 int w = g[u][k].second;
16                 d[v] = min(d[v], d[u] + w);
17             }
18         }
```

Implementación con optimización de espacio.

```
19      }
20
21      for (int u = 0; u < n; ++u){
22          for (int k = 0; k < g[u].size(); ++k){
23              int v = g[u][k].first;
24              int w = g[u][k].second;
25              if (d[v] > d[u] + w){
26                  // Se mejora la solucion agregando una arista
27                  return true;
28              }
29          }
30      }
31      // No hubo ciclo de peso negativo
32      // La distancia mas corta de s a u esta almacenada en d[u]
33      return false;
34  }
```

Resumen de algoritmos para SSSP

Algoritmo	**Uso**	**Complejidad**
BFS	Grafos sin pesos	$O(\|V\| + \|E\|)$
Dijkstra	Grafos con pesos ≥ 0	$O((\|V\| + \|E\|)\log \|V\|)$
Bellman-Ford	Grafos generales	$O(\|V\| \cdot \|E\|)$

Además el algoritmo de Bellman-Ford sirve para:

- Detectar si hay ciclos de peso negativo
- Hallar el camino más corto en cualquier grafo usando máximo i aristas

Algoritmo de Floyd-Warshall.

- Resuelve el “All Pair Shortest Path”.
- Entrada: un grafo G,
- objetivo: Hallas la distancia mas corta entre todos los pares de nodos, en caso de que lo haya, decir que hay un ciclo de peso negativo.

Generalidades

- Usa programación dinámica para hallar la distancia más corta entre todos los pares de nodos de un grafo con pesos.

- para propósitos del problema se va a considerar que los nodos están numerados desde 1 hasta n.

Algoritmo

Sea $d(i, j, k)$ la distancia más corta entre el nodo i el nodo j si como nodos intermedios solo se pueden usar los nodos $1 \ldots k$.

$$d(i,j,0) = \left\{ \begin{array}{ll} 0 & \text{si } i = j \\ C_{i,j} & \text{si } (i,j) \in E \\ +\infty & \text{en otro caso} \end{array} \right\} \text{ para } \begin{array}{l} 1 \leq i \leq n \\ 1 \leq j \leq n \end{array}$$

Donde $C_{i,j}$ es el costo que tiene asociado la arista (i, j)

En notación para DP.

$$d(i,j,k) = \min \begin{cases} d(i,j,k-1) & (\text{no usar nodo } k) \\ d(i,k,k-1) + d(k,j,k-1) & (\text{usar } k) \end{cases}$$

Para $1 \leq i,j,k \leq n$

Implementación.

```
1  for (int i = 1; i <= n; ++i) {
2     for (int j = 1; j <= n; ++j){
3        d[i][j][0] = INF;
4     }
5     d[i][i][0] = 0;
6  }
7
8  for (int edge = 0; edge < m; ++edge){
9     int u, v, c; cin >> u >> v >> c;
10    d[u][v][0] = min(d[u][v], c); //Por si hay un loop de peso > 0
11 }
12
13 for (int k = 1; k <= n; ++k){
14    for (int i = 1; i <= n; ++i){
15       for (int j = 1; j <= n; ++j){
16          d[i][j][k] = min(d[i][j][k-1],
17                           d[i][k][k-1] + d[k][j][k-1]);
18       }
19    }
20 }
```

Con optimización de espacio.

```
1  for (int i = 1; i <= n; ++i) {
2     for (int j = 1; j <= n; ++j){
3        d[i][j] = INF;
4     }
5     d[i][i] = 0;
6  }
7
8  for (int edge = 0; edge < m; ++edge){
9     int u, v, c; cin >> u >> v >> c;
10    d[u][v] = min(d[u][v], c); // Por si hay un loop de peso > 0
11 }
12
13 for (int k = 1; k <= n; ++k){
14    for (int i = 1; i <= n; ++i){
15       for (int j = 1; j <= n; ++j){
16          d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
17       }
18    }
19 }
```

Detección de ciclo negativo.

- Un grafo tiene un ciclo negativo si y solo si después de la ejecución de Floyd-Warshall, la diagonal principal contiene un elemento negativo.

Demostración

$\rightarrow$

Supongamos que hay un ciclo de peso negativo que empieza en el nodo i. Sea j el nodo más grande que pertenece a ese ciclo. En la ejecución del algoritmo cuando $k = j$
`d[i][i] = min(d[i][i], d[i][j] + d[j][i])`
Pero `d[i][j] + d[j][i]` es el costo del ciclo (< 0) ya que el mayor nodo del ciclo es el nodo k luego `d[i][i]` < 0.

$\leftarrow$

Supongamos que `d[i][i]` < 0 para algún i luego de la ejecución del algoritmo. Claramente hay un ciclo de peso negativo que contiene el nodo i.

Otras aplicaciones

- Determinar si existe un camino desde i hasta j. esto ocurre si al menos una de las siguientes afirmaciones ocurre.

- El nodo i es el nodo j.

- hay una arista del nodo i al nodo j.

- Existe un nodo k tal que hay un camino de i a k y de k a j.

Implementación

```
1  for (int i = 1; i <= n; ++i) {
2      for (int j = 1; j <= n; ++j){
3          d[i][j] = false;
4      }
5      d[i][i] = true;
6  }
7
8  for (int edge = 0; edge < m; ++edge){
9      int u, v; cin >> u >> v;
10     d[u][v] = true;
11 }
12
13 for (int k = 1; k <= n; ++k){
14     for (int i = 1; i <= n; ++i){
15         for (int j = 1; j <= n; ++j){
16             d[i][j] = d[i][j] or (d[i][k] and d[k][j]);
17         }
18     }
19 }
```

Otras aplicaciones.

- Minmax: Para cada pareja de nodos (i,j) , hallar el camino de i hasta j donde la arista m´as grande del camino sea lo más pequeña posible.

Abstracción

Sea $d(i,j,k)$ el costo de la arista más pequeña entre las aristas más grandes de todos los caminos de i hasta j si como nodos intermedios solo se pueden usar los nodos $1 \ldots k$.

$$d(i,j,0) = \left\{ \begin{array}{ll} 0 & \text{si } i = j \\ C_{i,j} & \text{si } (i,j) \in E \\ +\infty & \text{en otro caso} \end{array} \right\} \text{ para } \begin{array}{l} 1 \leq i \leq n \\ 1 \leq j \leq n \end{array}$$

$$d(i,j,k) = \min \left\{ \begin{array}{l} d(i,j,k-1) \\ \max \{d(i,k,k-1), d(k,j,k-1)\} \end{array} \right\}$$

$$\text{Para } 1 \leq i,j,k \leq n$$

Implementación

```
1  for (int i = 1; i <= n; ++i) {
2     for (int j = 1; j <= n; ++j){
3        d[i][j] = INF;
4     }
5     d[i][i] = 0;
6  }
7
8  for (int edge = 0; edge < m; ++edge){
9     int u, v, c; cin >> u >> v >> c;
10    d[u][v] = c;
11 }
12
13 for (int k = 1; k <= n; ++k){
14    for (int i = 1; i <= n; ++i){
15       for (int j = 1; j <= n; ++j){
16          d[i][j] = min( d[i][j] , max(d[i][k], d[k][j]) );
17       }
18    }
19 }
```

Otra aplicación

- Maximin: Para cada pareja de nodos (i,j) , hallar el camino de i hasta j donde la arista más pequeña del camino sea lo más grande posible.

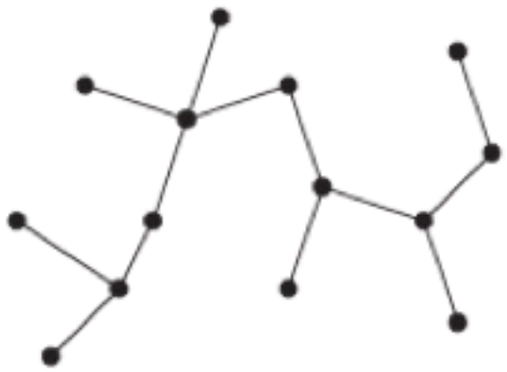
Implementación

```
1  for (int i = 1; i <= n; ++i) {
2     for (int j = 1; j <= n; ++j){
3        d[i][j] = -INF;
4     }
5     d[i][i] = INF;
6  }
7
8  for (int edge = 0; edge < m; ++edge){
9     int u, v, c; cin >> u >> v >> c;
10    d[u][v] = c;
11 }
12
13 for (int k = 1; k <= n; ++k){
14    for (int i = 1; i <= n; ++i){
15       for (int j = 1; j <= n; ++j){
16          d[i][j] = max( d[i][j] , min(d[i][k], d[k][j]) );
17       }
18    }
19 }
```

Árbol de mínima expansión.

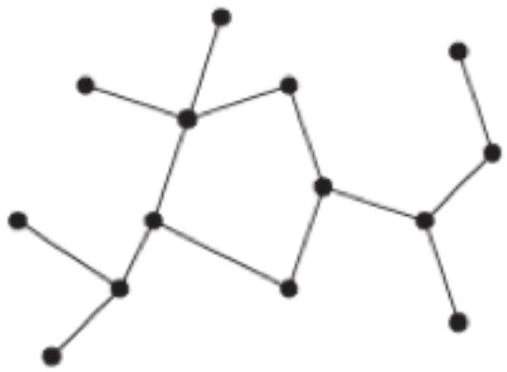
Árboles

Un árbol es un grafo **no dirigido, conexo, y no cíclico**.
Un grafo no dirigido y no cíclico pero no necesariamente conexo (todos los nodos están conectados) es un bosque.



(a)

(b)



(c)

Curiosidad de los árboles… propiedad

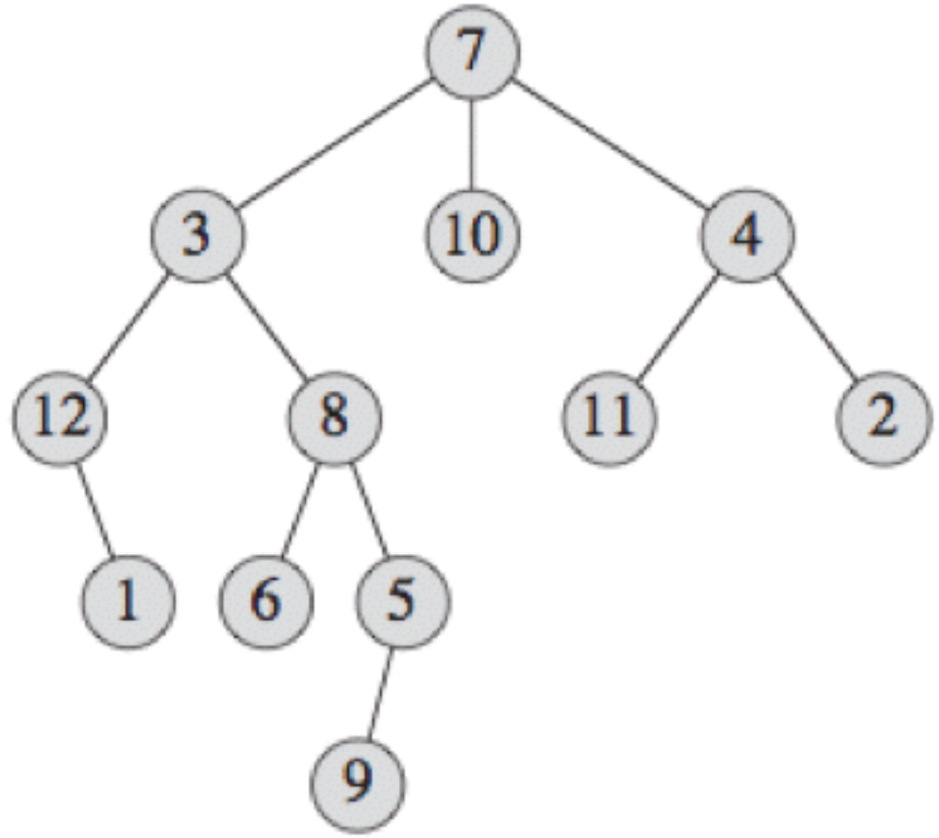
Propiedades

Sea $G = (V, E)$ un grafo no dirigido. Los siguientes enunciados son equivalentes:

- G es un árbol
- Hay un único camino entre cualquier par de nodos $u, v \in V$
- G es un grafo conexo y $|E| = |V| - 1$

Árbol con raíz

- un árbol T(V,E) tiene raíz cuando se decide diferenciar uno de sus nodos, el nodo que se diferencia de los demás se llama raíz y se simboliza por r.



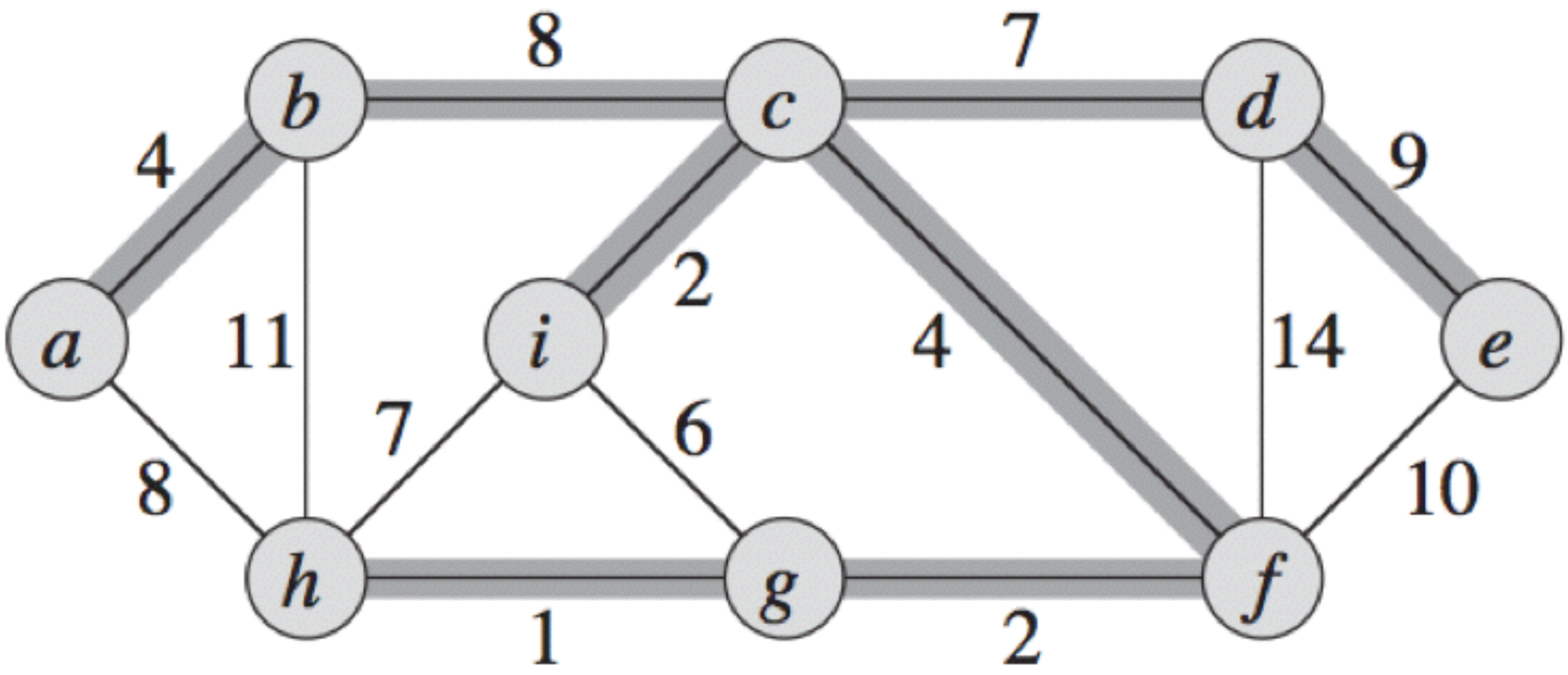
Árbol de mínima expansión

- Entrada: Un grafo G no dirigido y conexo, una función de pesos W(u,v) donde u y v pertenecen a E.

- Objetivo: hallar un conjunto no cíclico T subconjunto de E que conecte todos los nodos de G y que minimice su peso. el peso de T es la suma de todos los W(u,v) con u y v pertenecientes a E.

- ya que T es no cíclico, conexo y no dirigido, es un árbol, y a éste se le llama de mínima expansión.

Ejemplo



Algoritmo de Prim.

- Seleccionar un nodo cualquiera y colocarlo en la lista de visitados.

- mientras que haya nodos sin visitar:

- tomar la arista con menor peso que conecte un nodo visitado u y uno no visitado v.

- agregar la arista (u,v) al MST

- agregar v al conjunto de visitados.

- FinMientras.

Optimización

- En este algoritmo repetidas veces se extrae el menor de una lista.

- Esto se puede hacer mas rápido utilizando un HEAP, como se hizo en el algoritmo de Dijkstra.

Implementación

```
1  const int MAXN = 10005;
2  typedef pair <int, int> edge;
3  bool visited[MAXN];
4  // g[i] = lista de parejas (nodo, peso)
5  vector <pair <int, int> > g[MAXN];
6
7  int prim(int n){
8      for (int i = 0; i <= n; ++i) visited[i] = false;
9      int total = 0;
10     // Crear el heap de forma que se extraiga el de menor peso
11     // El heap es de parejas (peso, nodo), contrario al grafo
12     priority_queue<edge, vector <edge>, greater<edge> > q;
13     // Empezar el MST desde el nodo 0
14     q.push(edge(0, 0));
15     while (!q.empty()){
16         int u = q.top().second;
17         int w = q.top().first;
18         q.pop();
```

Implementación

```
19          // Si es una arista entre dos nodos ya incluidos al MST
20          if (visited[u]) continue;
21
22          visited[u] = true;
23          total += w;
24          for (int i = 0; i < g[u].size(); ++i){
25              int v = g[u][i].first;
26              int next_w = g[u][i].second;
27              // Si v no pertenece todavia al MST
28              if (!visited[v]){
29                  // Insertar primero el peso y luego el nodo
30                  q.push(edge(next_w, v));
31              }
32          }
33      }
34      return total; // El costo total del MST
35  }
```

Estructura Union-find.

Union-Find es una estructura de datos para almacenar una colección conjuntos disjuntos (no tienen elementos en común) que cambian dinámicamente.

Para hacer esto identifica en cada conjunto un “padre” que es un elemento al azar de ese conjunto y hace que todos los elementos del conjunto “apunten” hacia ese padre.

Inicialmente se tiene una colección donde cada elemento es un conjunto unitario.

Estructura union-find

Esta estructura de datos tiene dos operaciones:

Union(u, v) Une los conjuntos que contienen a u y a v en un solo conjunto.

Find(u) Retorna el padre del (único) conjunto que contiene a u.

Implementación

```
1  int p[MAXN]; //p[i] el padre del conjunto al que pertenece i
2
3  // Inicializa los conjuntos de los elementos 0 a n
4  void initialize(int n){
5      for (int i = 0; i <= n; ++i) p[i] = i;
6  }
7  // Retorna el padre del conjunto que contiene a u
8  int find(int u){
9      if (p[u] == u) return u;
10     return p[u] = find(p[u]);
11 }
12 // Une los conjuntos a los que pertenecen u y v
13 // Este nuevo conjunto tiene como padre el padre de v
14 void join(int u, int v){
15      int a = find(u);
16      int b = find(v);
17      if (a == b) return; // Son el mismo conjunto
18      p[a] = b; // El padre del padre de u es el padre de v
19 }
```

Algoritmo de Kruskall.

- Usa union-find para verificar que al agregar una arista no se este generando un ciclo.

- Ordenar las aristas ascendentemente -> desempate es indiferente.

- para cada arista en ese ordenamiento:

- si agregar la arista no genera ciclo:

- agregar la arista al MST

- finSi

- FinPara

Implementación

```
1  // Estrutura personalizada para manejar las aristas.
2  // En el algoritmo de Kruskal el grafo se guarda como lista de
       aristas y no como lista de adyacencia
3  struct edge{
4      // Atributos
5      int start, end, weight;
6      // Constructor
7      edge(int u, int v, int w){
8          start = u; end = v; weight = w;
9      }
10     // Comparador menor
11     bool operator < (const edge &other) const{
12         return weight < other.weight;
13     }
14 }; // No olvidar este ;
```

Implementación

```
18  const int MAXN = 100005;
19  vector <edge> edges;
20  int p[MAXN];
21
22  int find(int u){
23      if (p[u] == u) return u;
24      return p[u] = find(p[u]);
25  }
26
27  void join(int u, int v){
28      int a = find(u);
29      int b = find(v);
30      if (a == b) return;
31      p[a] = b;
32  }
```

Implementación

```
37  int kruskal(int n){
38      // Inicializar el arreglo de union-find
39      for (int i = 0; i <= n; ++i) p[i] = i;
40      // Ordenar las aristas de menor a mayor
41      sort(edges.begin(), edges.end());
42
43      int total = 0;
44      for (int i = 0; i < edges.size(); ++i){
45          int u = edges[i].start;
46          int v = edges[i].end;
47          int w = edges[i].weight;
48          if (find(u) != find(v)){ // Si no genera un ciclo
49              total += w;
50              join(u, v);
51          }
52      }
53      return total;
54  }
```

Algoritmo de flujo máximo.

- ¿cuál es la tasa mayor a la cual el material puede ser transportado de la fuente al sumidero sin violar ninguna restricción de capacidad?

- En otras palabras, el problema consiste en determinar la máxima capacidad de flujo que puede ingresar a través de la fuente y salir por el nodo de destino.

Red de flujos

Red de flujos

Una red de flujos $G = (V, E)$ es un grafo conexo y dirigido en donde cada arista $(u, v) \in E$ tiene una capacidad **no negativa** $c(u, v) \geq 0$.

Si $(u, v) \notin E$ entonces $c(u, v) = 0$

Se distinguen dos nodos: la fuente s y el sumidero t.

Flujo

Sea $G = (V, E)$ una red de flujos con fuente s y sumidero t.

Flujos

Un flujo es una función $f : V \times V \to \mathbb{R}$ que cumple que:

- **Restricción de capacidad:**

$$f(u, v) \leq c(u, v) \quad \forall\, u, v \in V$$

- **Simetría:**

$$f(u, v) = -f(v, u) \quad \forall\, u, v \in V$$

- **Conservación de flujo:** El flujo que sale de un nodo diferente de s y t es igual al que entra al nodo.
  $\forall\, u \in V - \{s, t\}$

$$\sum_{v \in V} f(u, v) = \sum_{\substack{v \in V \\ f(u,v)<0}} f(u, v) + \sum_{\substack{v \in V \\ f(u,v)>0}} f(u, v) = 0$$

Problema de flujo máximo

Sea $G = (V, E)$ una red de flujos con fuente s, sumidero t, función de flujos f.

Definiciones

- Al valor $f(u, v)$ se le llama flujo del nodo u al nodo v.
- El valor del flujo de G se denota por $|f|$ y corresponde al flujo que sale de s menos el flujo que entra a s

$$|f| = \sum_{v \in V} f(s, v)$$

Problema de máximo flujo

Dados G, s y t hallar un flujo de G que tenga valor máximo.

Red Residual

Dados una red e flujos $G = (V, E)$ y un flujo f.
La red residual de G dado que se ha "bombeado" un flujo f es un grafo $G_f = (V, E_f)$ con una función de costos c_f.
El peso c_f de una arista $(u, v) \in E_f$ es el valor de flujo que todavía se puede "bombear" desde u hasta v sin violar la capacidad de (u, v).

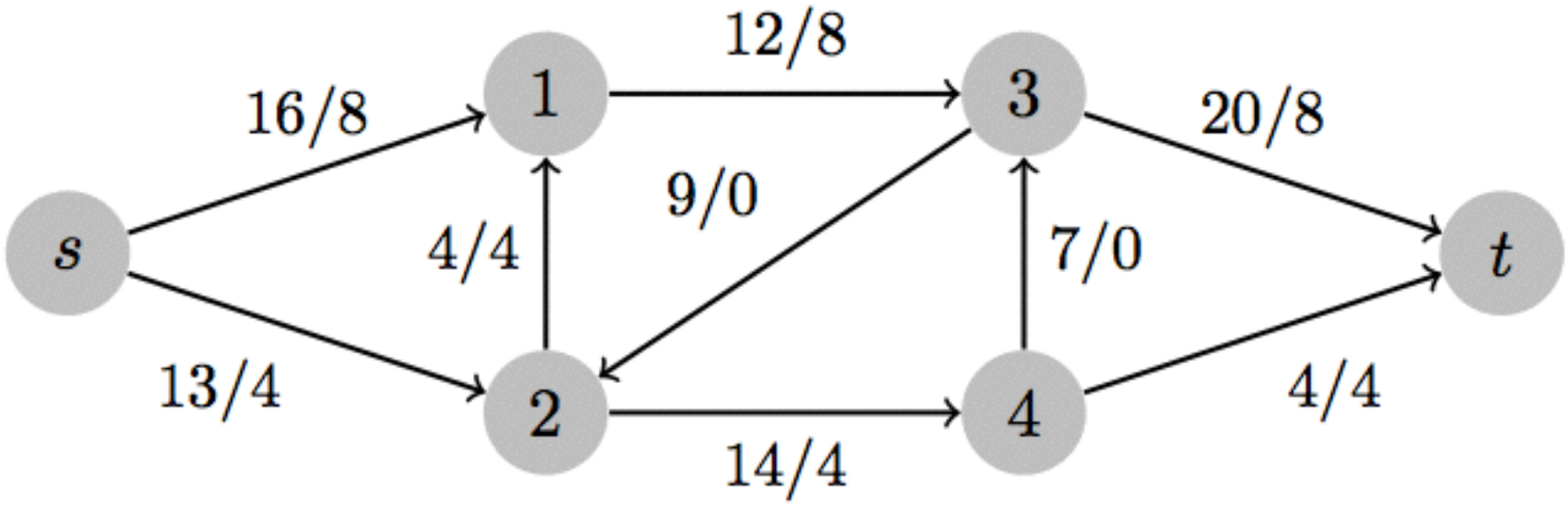
$$c_f(u, v) = c(u, v) - f(u, v)$$

Ejemplo

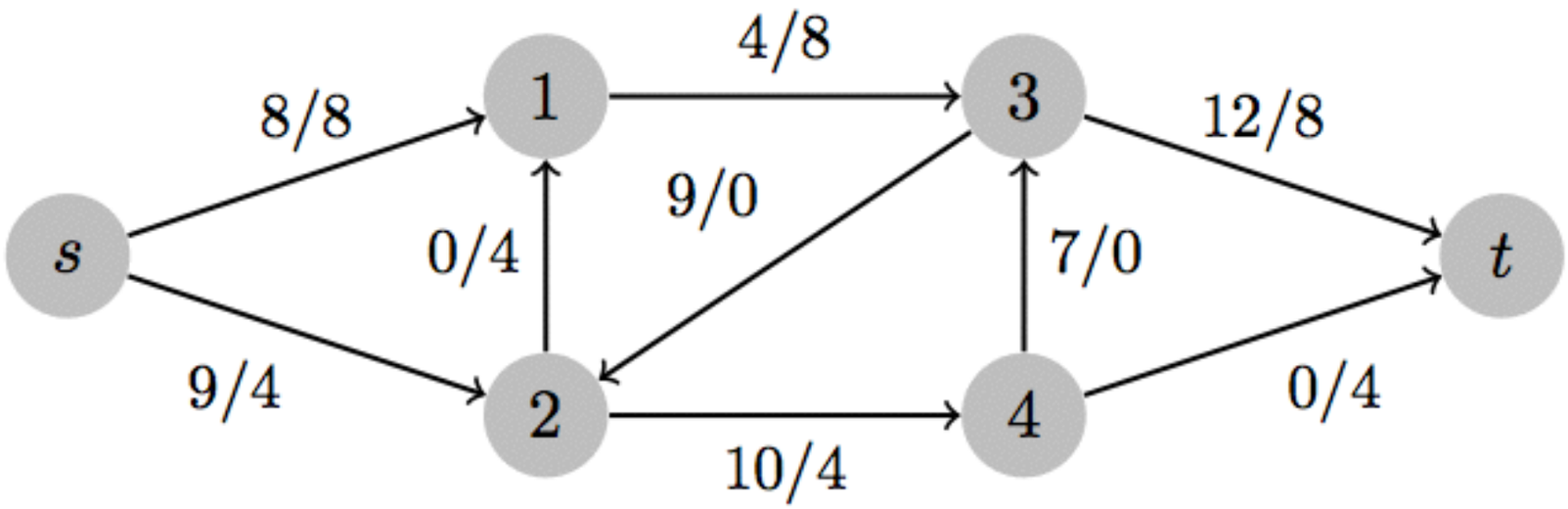
- Si $c(u,v) = 16$ y $f(u,v) = 11$ entonces $c_f(u,v) = 16 - 11 = 5$. Esto significa que por la arista (u,v) todavía se pueden bombear 5 unidades de flujo sin violar la capacidad.
- Si $c(u,v) = 16$ y $f(u,v) = -4$ (es decir que $f(v,u) = 4$) entonces $c_f(u,v) = 16 - (-4) = 20$. Esto significa que por la arista (u,v) no sólo se pueden "bombear" las 16 unidades que tiene de capacidad esa arista, sino que también se pueden "desbombear" las 4 unidades que se habían bombeado de v a u.

Ejemplo

Red de flujos: Peso de arista (a, b) es $c(a, b)/f(a, b)$



Red residual: Peso de arista (a, b) es $c_f(a, b)/c_f(b, a)$



Camino de aumentación

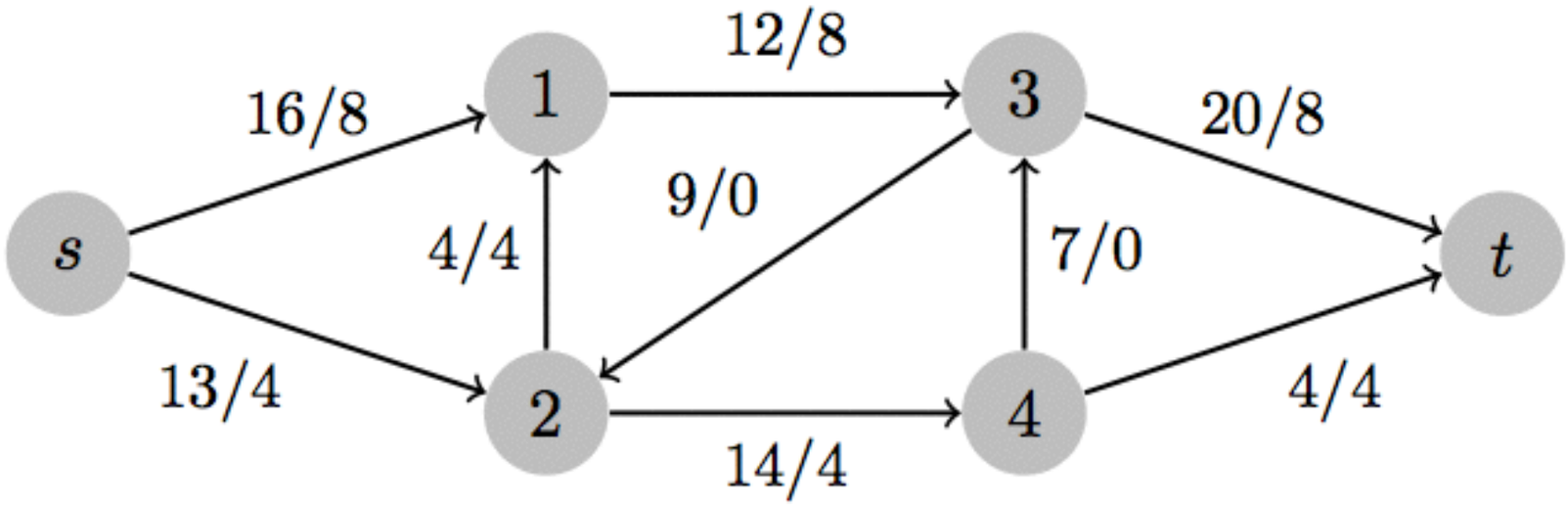
Camino de aumentación

Dada una red residual G_f un camino de aumentación es un camino simple (sin ciclos) p que va de s a t en G_f tomando solo aristas de peso mayor que 0.

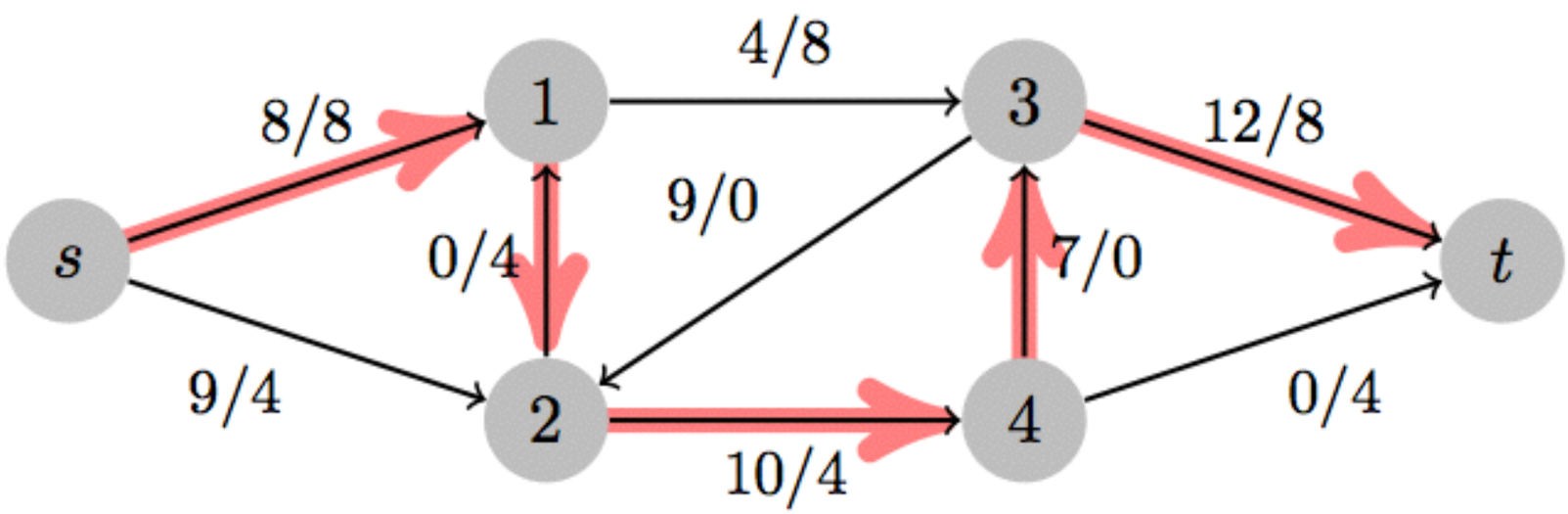
En otras palabras un camino de aumentación es un camino por el cual todavía se puede bombear flujo de s a t en G sin violar las restricciones de capacidad.

Ejemplo

Red de flujos: Valor escrito en arista (a, b) es $c(a, b)/f(a, b)$



Red residual: Valor escrito en arista (a, b) es $c_f(a, b)/c_f(b, a)$



Cuello de botella

La máxima cantidad de flujo que se puede enviar por un camino de aumentación p corresponde al mínimo valor de las aristas de p en la red residual.

Este valor se conoce como cuello de botella

$$\text{bottleneck}(p) = \min\{c_f(u, v) | (u, v) \in p\}$$

Aumentación de flujo

Se tienen una red de flujos $G = (V, E)$, un flujo f y un camino p en la red residual.
Si se aumenta el flujo $f(u, v)$ en el valor bottleneck(p) para cada $(u, v) \in p$ y se reduce en bottleneck(p) para (v, u) entonces:

- El valor resultante para $f(u, v)$ cumple con las propiedades de flujo.
- El valor del nuevo flujo es el valor del antiguo flujo más bottleneck(p).

Teorema

Max-flow min-cut

Si f es un flujo en una red de flujos $G = (V, E)$ con fuente s y sumidero t entonces las siguientes condiciones son equivalentes:

- f es un flujo máximo en G.
- La red residual G_f no contiene caminos de aumentación
- $|f|$ es el mínimo costo de cortar aristas de G de manera que s y t queden separados cuando costo de cortar una arista es el peso de la arista.

Algoritmo

1. Inicializar el flujo en 0.
2. Mientras que haya caminos de aumentación p en G_f
   3. Aumente el flujo f a lo largo de p en el valor bottleneck(p)
4. retornar $|f|$

Ford-Fulkerson.

El algoritmo de Ford-Fulkerson utiliza DFS para hallar el camino de aumentación.

1. Para todo u y v $f(u,v) = 0$.
2. $flow = 0$
3. Mientras que se encuentre un caminos de aumentación p en G_f usando DFS
   4. $bottleneck = \min\{c_f(u,v) : (u,v) \in p\}$
   5. Para cada $(u,v) \in p$
      6. $f(u,v) += bottleneck$
      7. $f(v,u) = -f(u,v)$
   8. $flow += bottleneck$
9. retornar $flow$

Ejemplo

Figura : Red de flujos

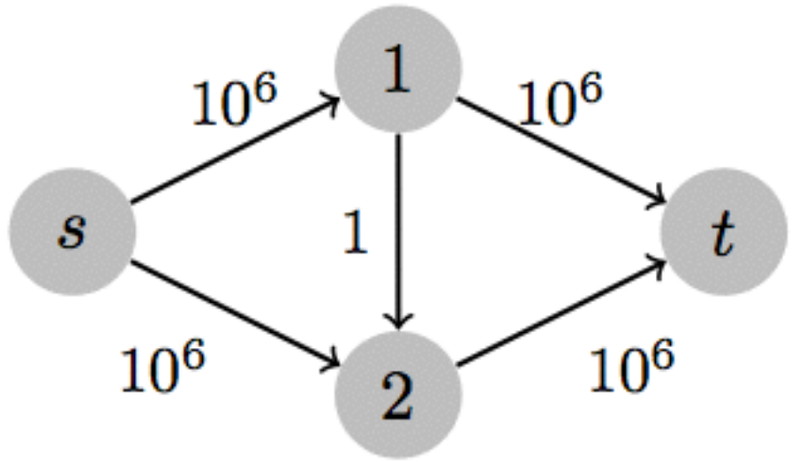


Figura : Red residual iteración 1

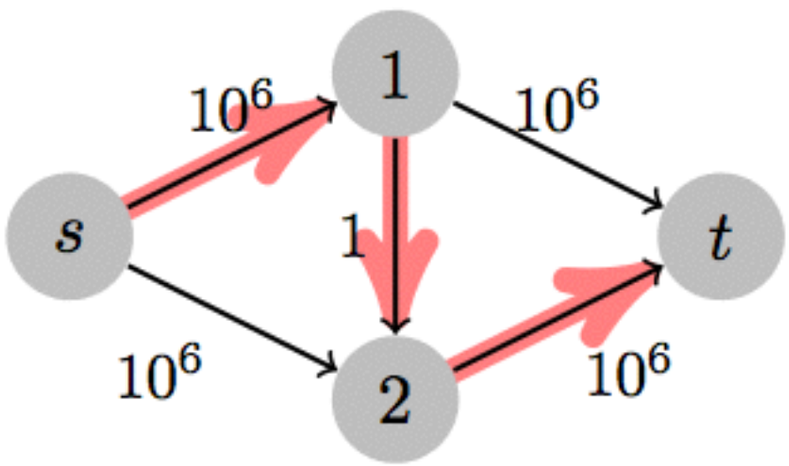
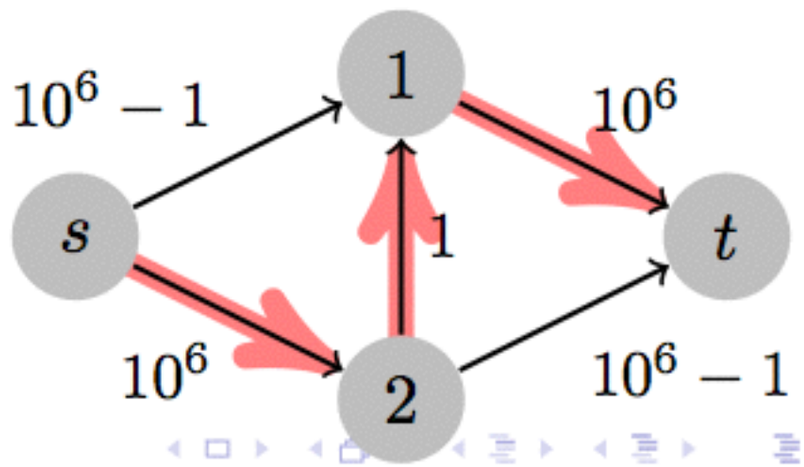


Figura : Red residual iteración 2



Nota

- Hay mas algoritmos que resuelven este ejercicio además de Ford-Fulkerson como el de Edmond-Karp, entre otros.