Grafos

Román Castellarin

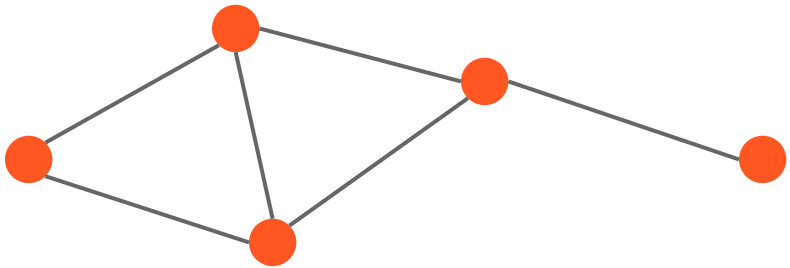
Definiciones básicas
Representación
DFS/BFS
Aplicaciones
Disjoint Sets
MST
Dijkstra
Floyd-Warshall

¿Qué son?

Puntitos y rayitas.

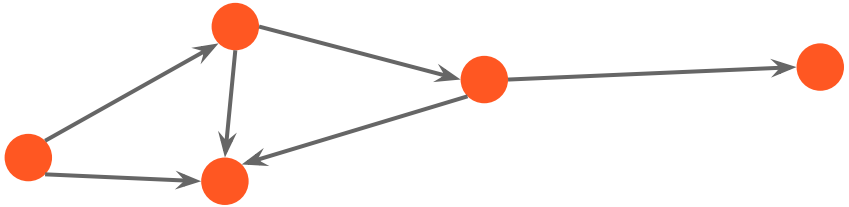
Más formalmente, un grafo G = (V, E) consiste de un conjunto V de vértices, y un conjunto E de aristas que conectan cada una dos vértices.

Generalmente llamamos N = |V| y M = |E|.

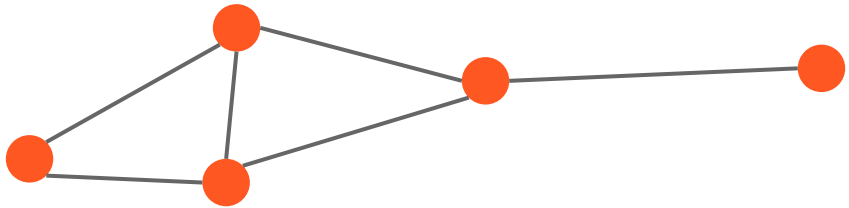


Grafos

Un grafo es **dirigido** si las aristas están orientadas.
(Generalmente las llamamos arcos).

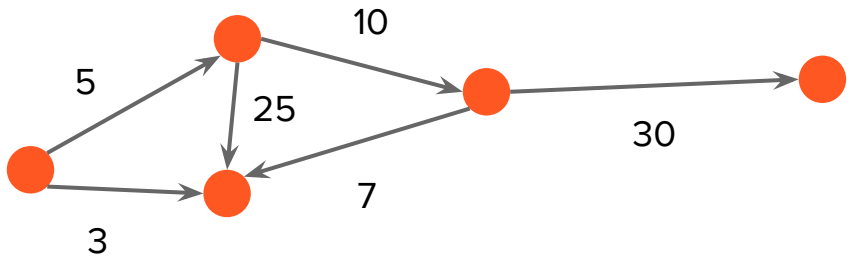


Es no dirigido si no importa la dirección.



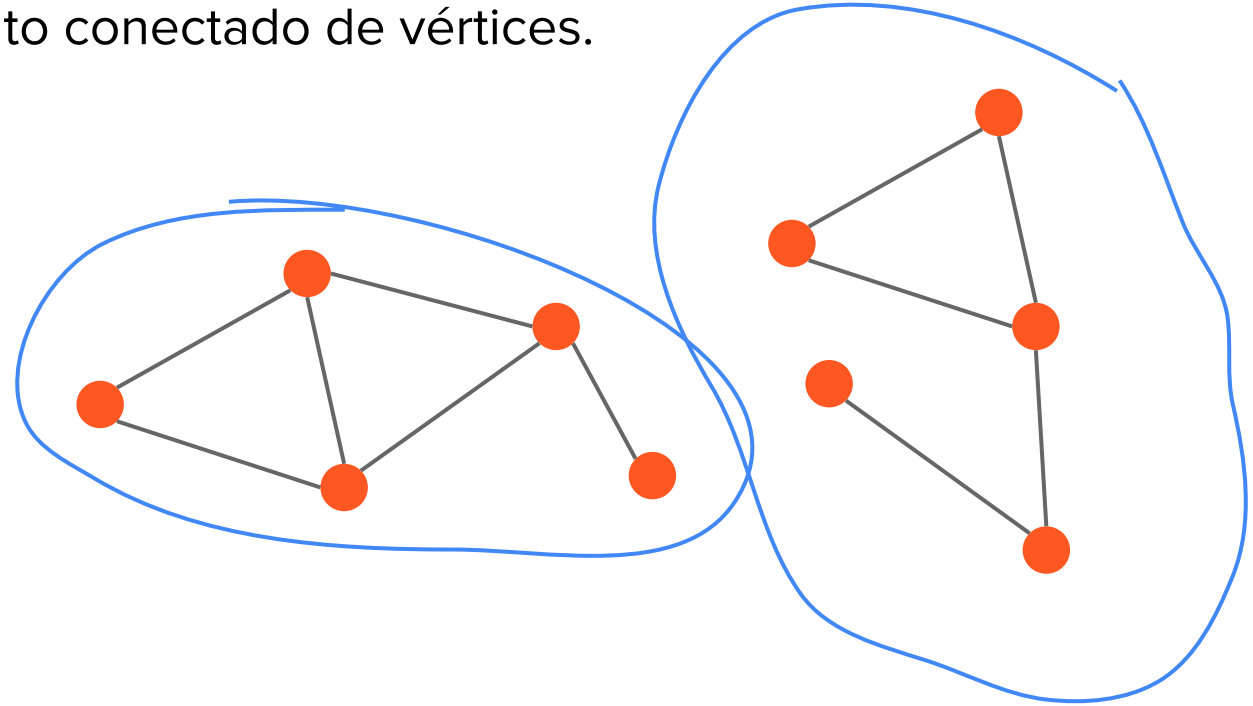
Grafos

Un grafo está **ponderado** si las aristas tienen pesos (etiquetas numéricas) asociados. Algunas veces aparecen pesos en los vértices.



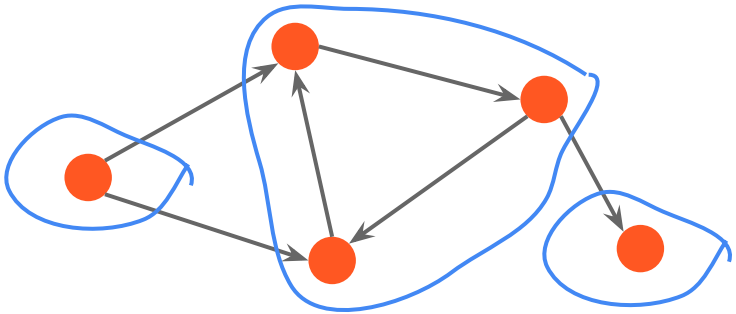
Grafos

En un grafo no dirigido, una **componente** (**débilmente**) **conexa** es un conjunto conectado de vértices.



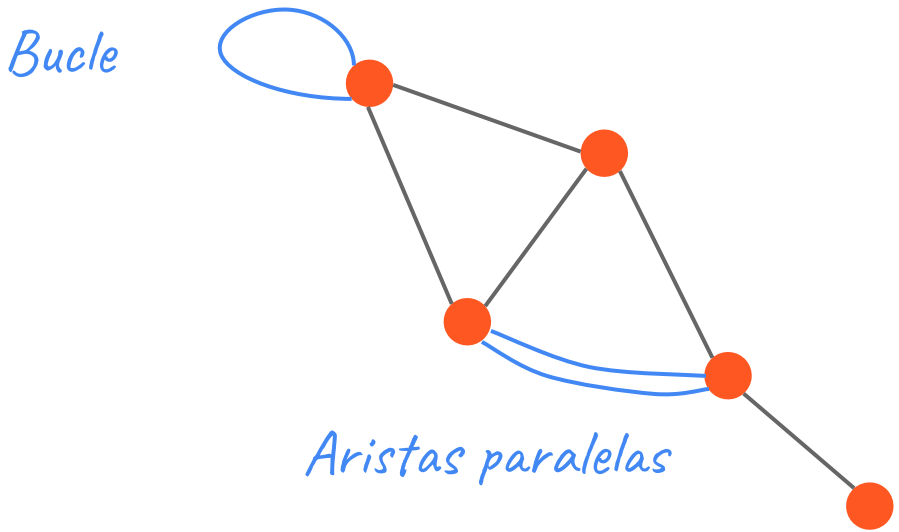
Grafos

Para los grafos dirigidos, además, una **componente fuertemente conexa** es un conjunto maximal de vértices tales que para todo par u, v en dicha componente, existe el camino u ⇝ v.



Grafos

Un grafo es **simple** si no tiene bucles ni aristas paralelas.



Ejemplos

Podemos representar una red social como un grafo no dirigido.
Cada vértice es una persona, y hay una arista entre u y v si son amigos.

Cada clique es un grupo de amigos.

Ejemplos

Podemos representar una red social como un grafo no dirigido.
Cada vértice es una persona, y hay una arista entre u y v si son amigos.

Los vértices de alto grado son influencers

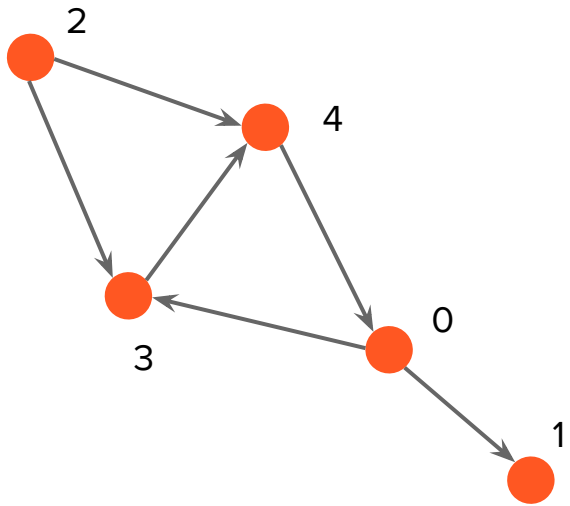
(el grado de un vértice es la cantidad de aristas que inciden en él)

Representación

Lista de Incidencias

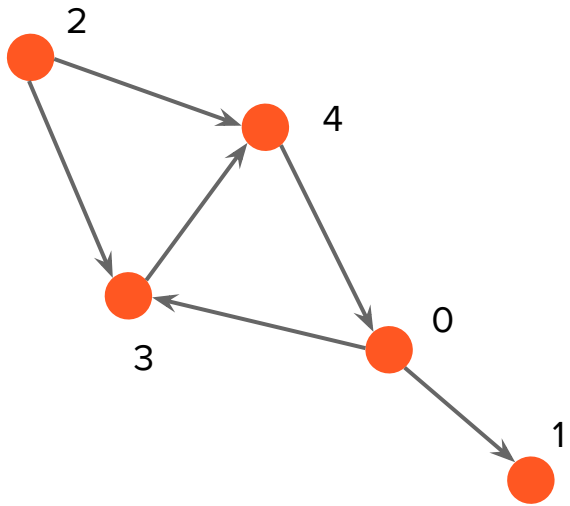
Es la forma más usual en la que aparecen los grafos en los archivos de **entrada**. Consiste en una simple lista de aristas, usualmente precedida de la cantidad de vértices y aristas.

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



Lista de Incidencias

```
struct edge{
   int a, b;
};
int N, M;
vector<edge> E;
…

cin>>N>>M;
forn(i, M){
     int a, b;
     cin >> a >> b;
     E.push_back({a,b});
}
```

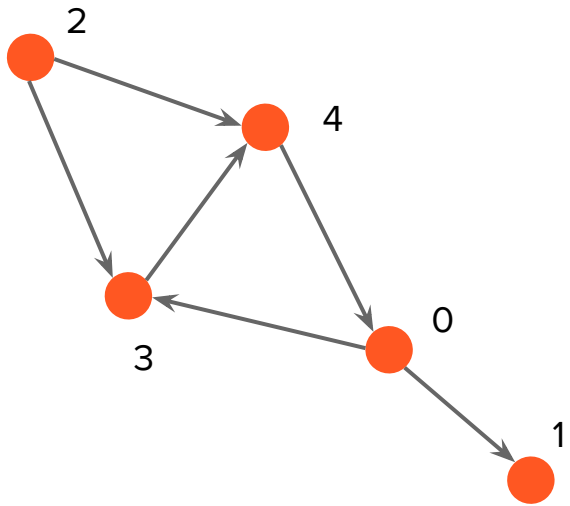


Matriz de Adyacencia

Consiste en una matriz A de NxN tal que `A[i][j] = 1` sii hay una arista (i, j)

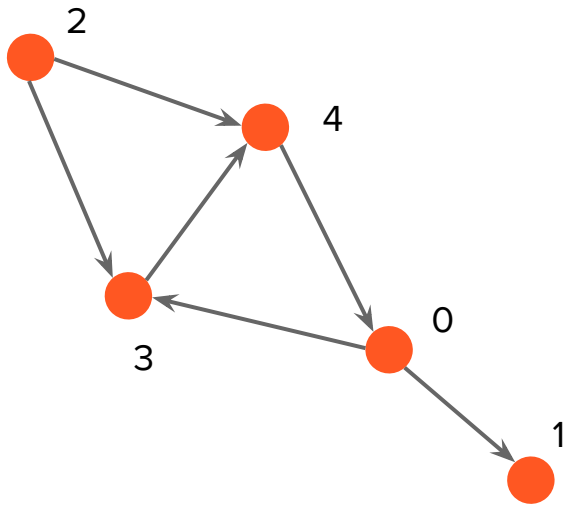
```
int A[MAXN][MAXN];
```

```
0 1 0 1 0
0 0 0 0 0
0 0 0 1 1
0 0 0 0 1
1 0 0 0 0
```



Matriz de Adyacencia

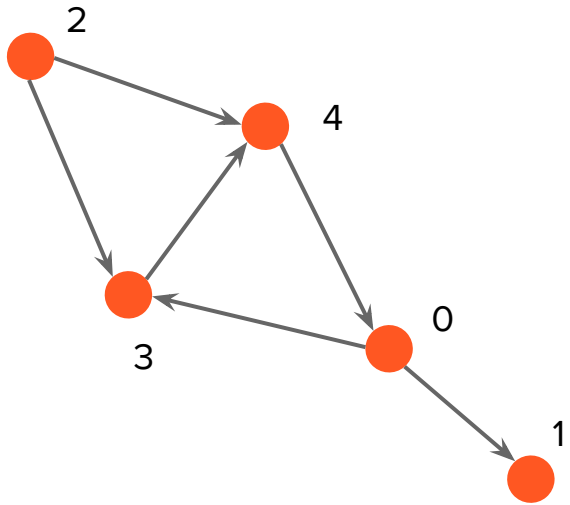
```
int N, M;
int A[MAXN][MAXN];
…

cin>>N>>M;
forn(i, M){
     int a, b;
     cin >> a >> b;
     A[a][b] = 1;
}
```



Lista de Adyacencias

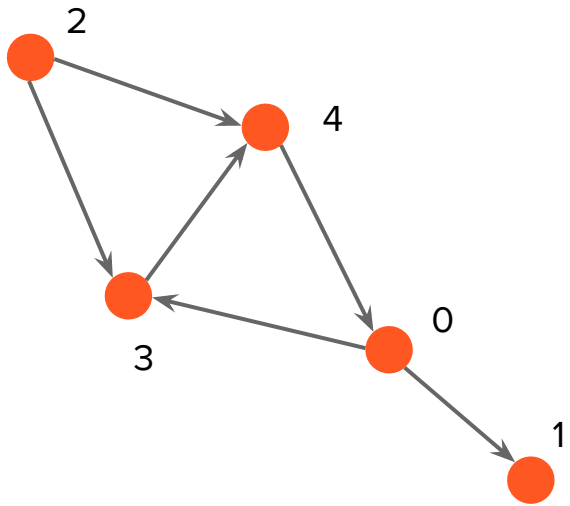
Es la forma preferida por los algoritmos de grafos más comunes.
Para cada vértice, guardar una lista de sus vecinos.

```
vector<int> G[MAXN];

0: 1, 3
1:
2: 3, 4
3: 4
4: 0
```



Lista de Adyacencias

```
int N, M;
vector<int> G[MAXN];
…

cin>>N>>M;
forn(i, M){
     int a, b;
     cin >> a >> b;
     G[a].push_back(b);
}
```

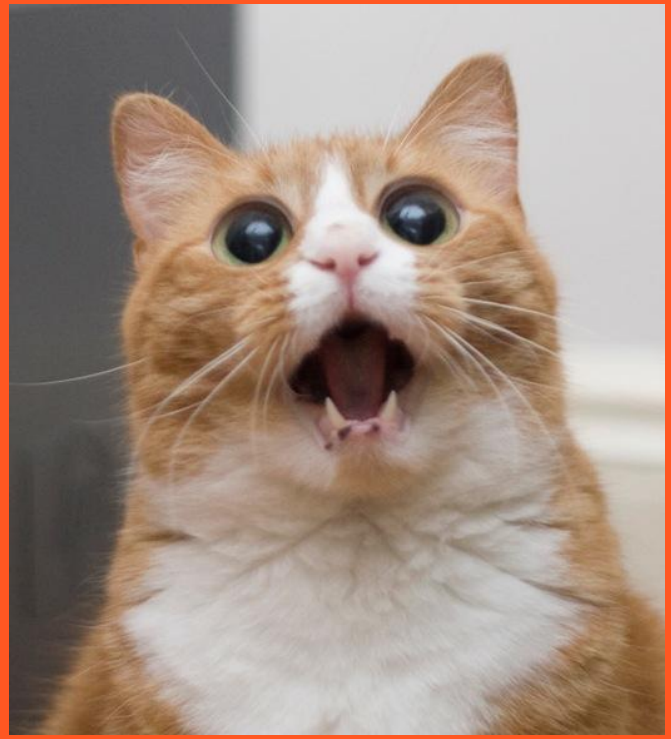


Representaciones

	Memoria	Iterar vecinos de v	Verificar si existe la arista u-v
Lista de Incidencias	O(M)	O(M)	O(M)
Matriz de Adyacencia	$O(N^2)$	O(N)	O(1)
Lista de Adyacencias	O(N+M)	$O(\delta v)$	$O(\min(\delta u, \delta v))$

δv : grado de v (cantidad de vecinos)

DFS/BFS



gato luego de aprender BFS

Caminos

Un **camino** u-w es una secuencia de vértices u = v_0, v_1, .., v_n= w adyacentes.

Un **ciclo** es un camino que comienza y termina en el mismo vértice.
La **longitud** de un camino es la cantidad de aristas en él.
Por lo tanto, todo bucle es un camino de longitud 1.

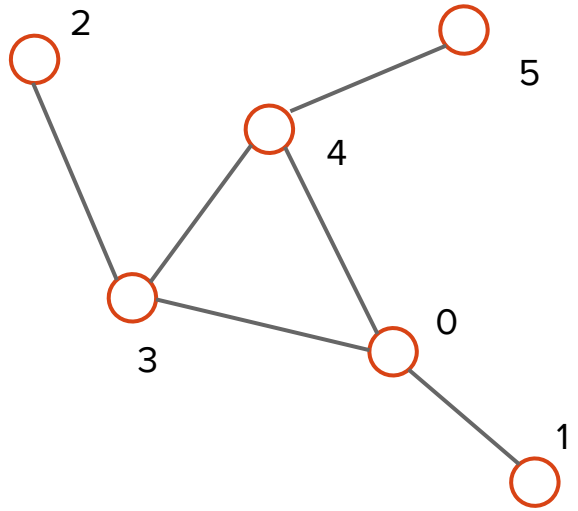
La **distancia** entre u y w es la longitud del menor camino u-w.

¿Cómo detectamos ciclos?
¿Cómo calculamos distancias?

DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

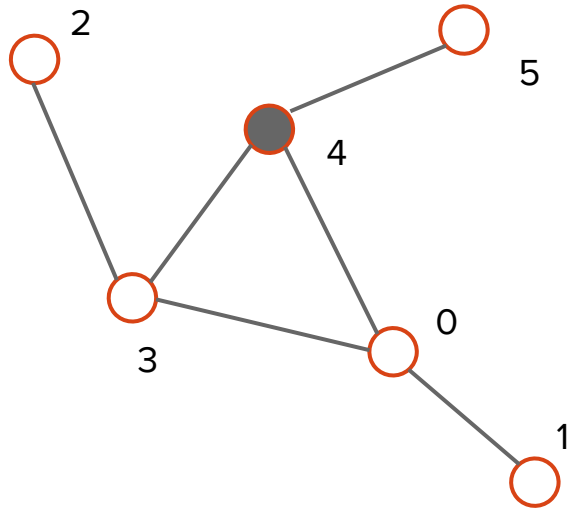
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

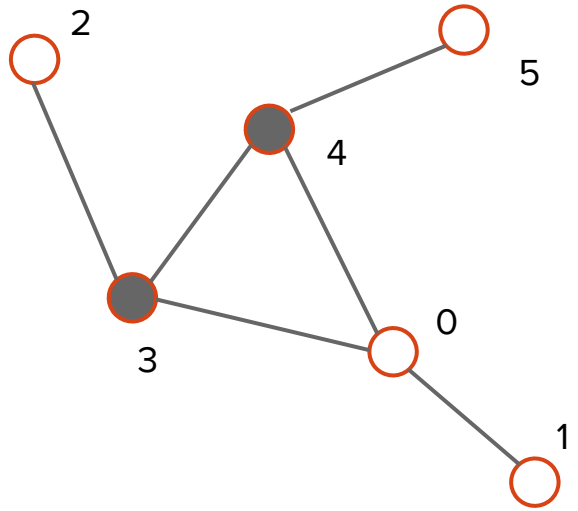
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

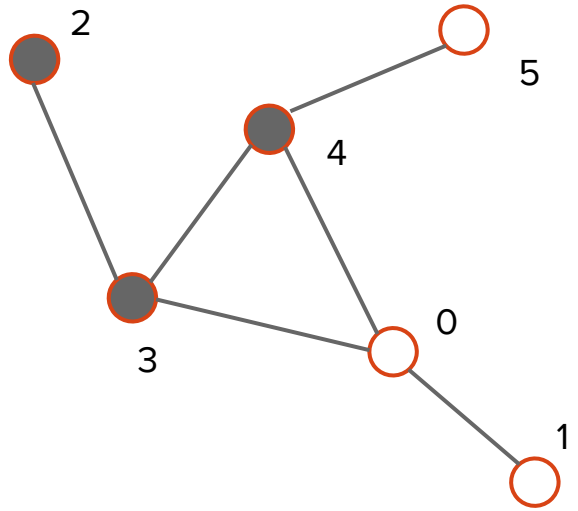
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

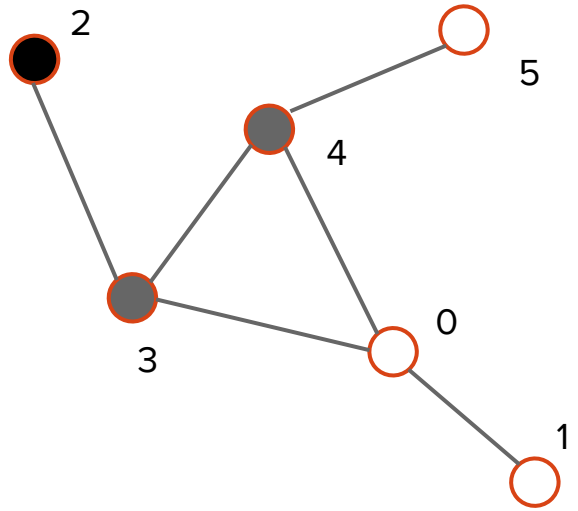
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

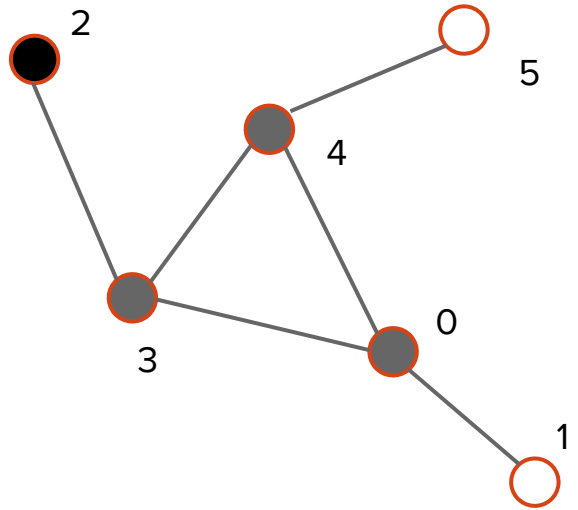
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

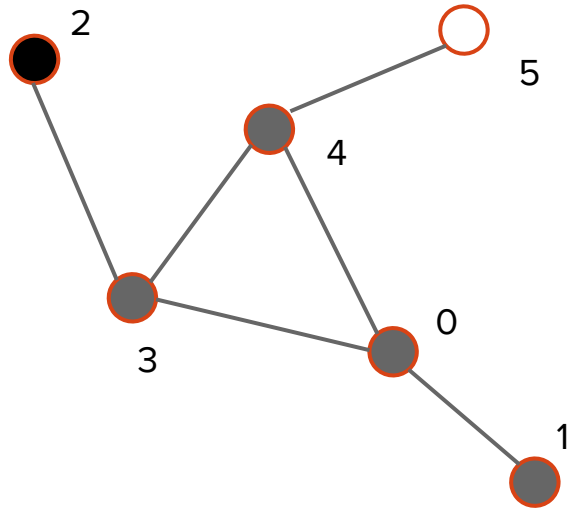
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

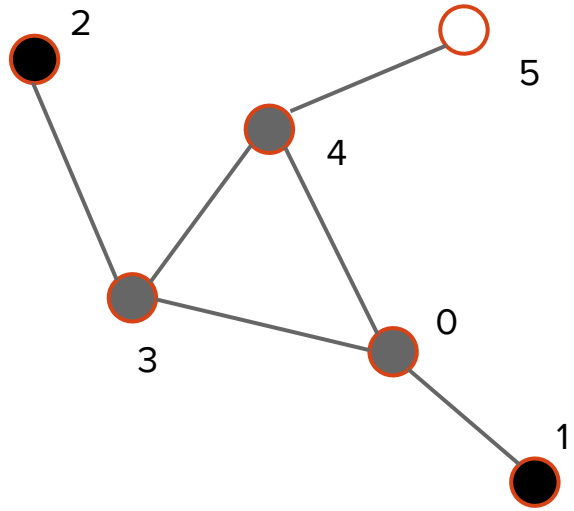
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

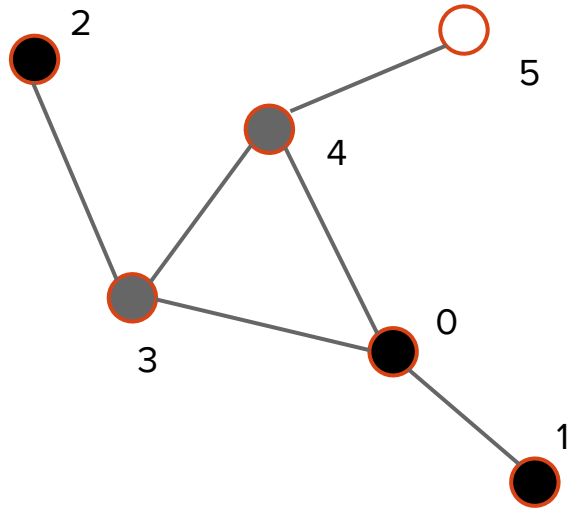
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

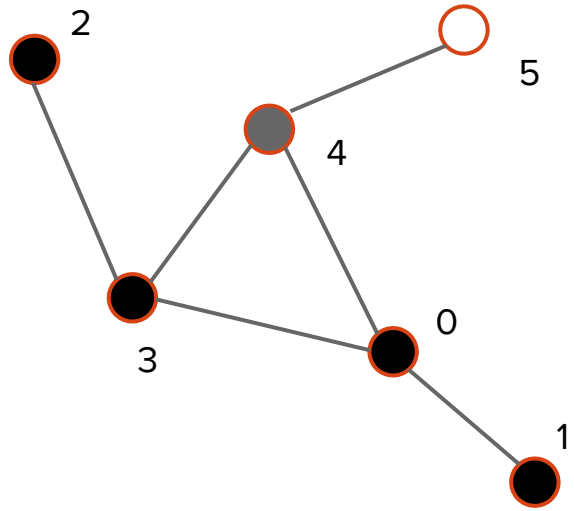
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

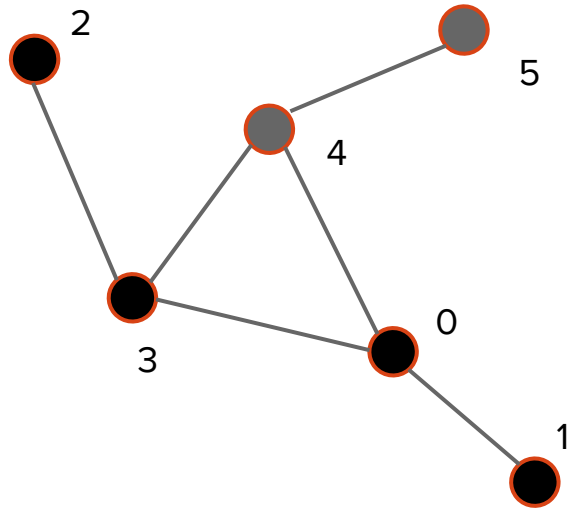
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

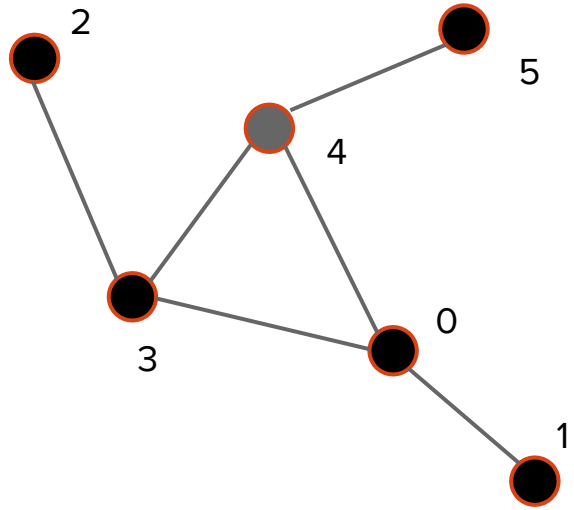
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

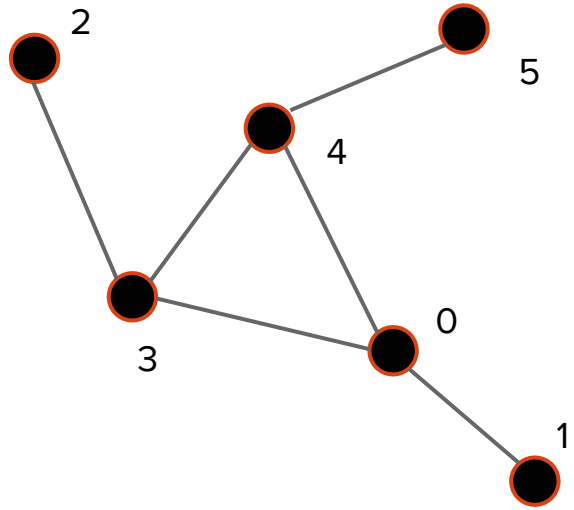
```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Un recorrido por profundidad (depth-first search) comienza en un vértice inicial y visita recursivamente todos los vértices no visitados.

```
void dfs(int v){
    color[v] = GRIS;
    for(auto &w: G[v])
        if( color[w] == BLANCO )
            dfs(w);
    color[v] = NEGRO;
}
```



DFS

Podemos detectar **ciclos** en un grafo **no dirigido** si encontramos un vecino no visitado que no sea mi padre (es decir, aquél de donde vine).

```
bool hasCycle(int v, int p=-1){
     visited[v] = true; // más fácil que guardar colores
     bool cycle = false;
     for(auto &w: G[v])
          if( not visited[w] )
               cycle |= hasCycle(w, v);
          else if( w != p )
               cycle = true;
     return cycle;
}
```

DFS

Podemos detectar **ciclos** en un grafo **dirigido** si encontramos una *back-edge* (arco a un vértice gris).

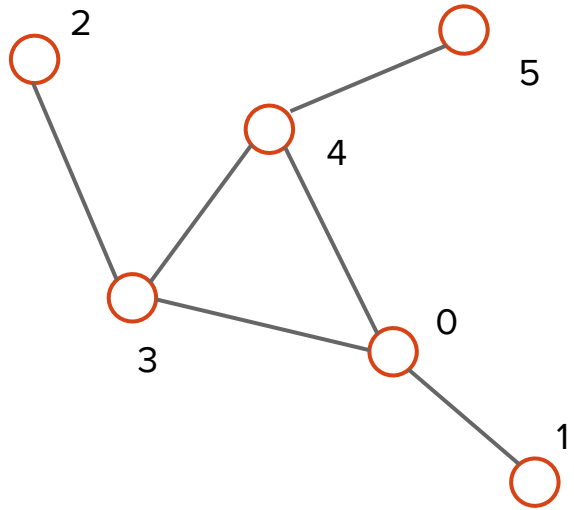
```
bool hasCycle(int v){
     color[v] = GRIS;
     bool cycle = false;
     for(auto &w: G[v])
          if( color[w] == BLANCO )
               cycle |= hasCycle(w);
          else if( color[w] == GRIS )
               cycle = true;
     return cycle;
}
```

BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

Entendido como cantidad de arcos

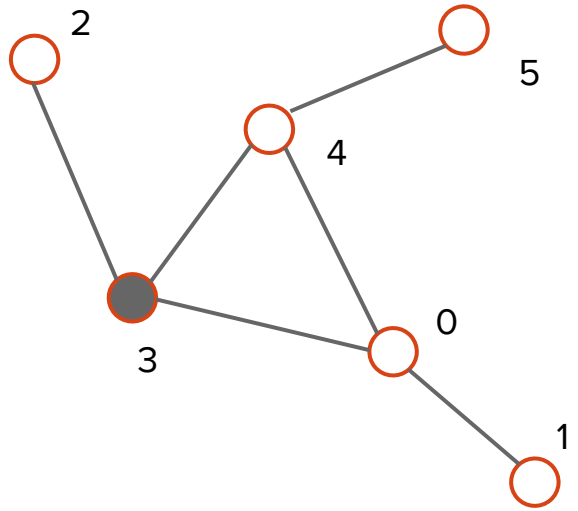
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

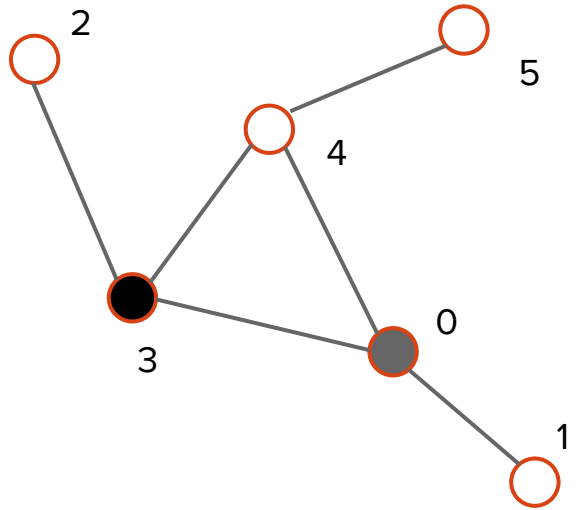
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

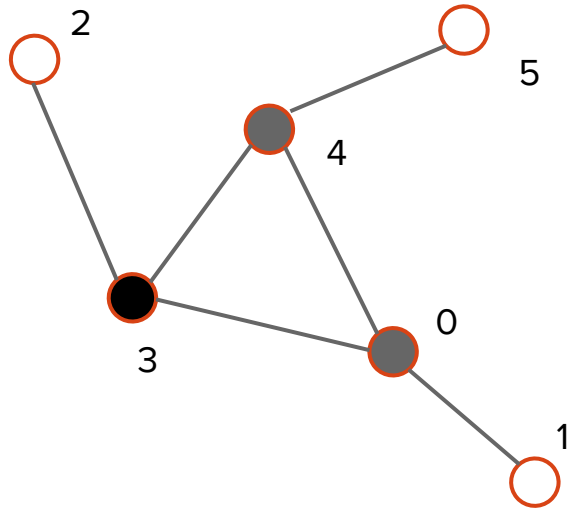
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

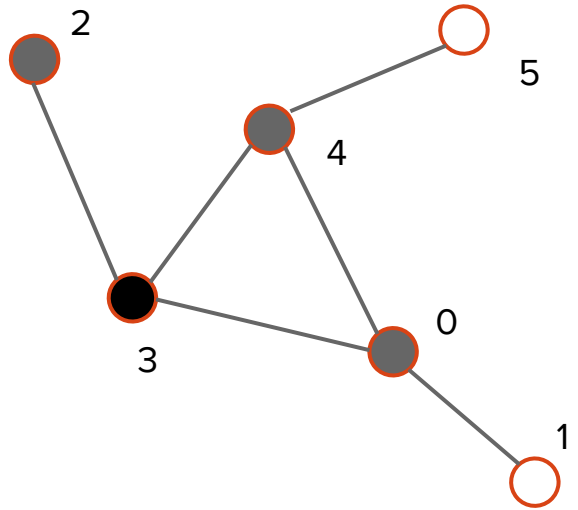
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

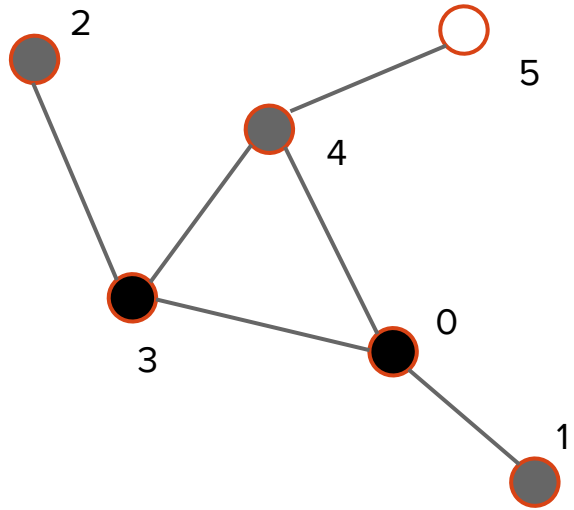
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

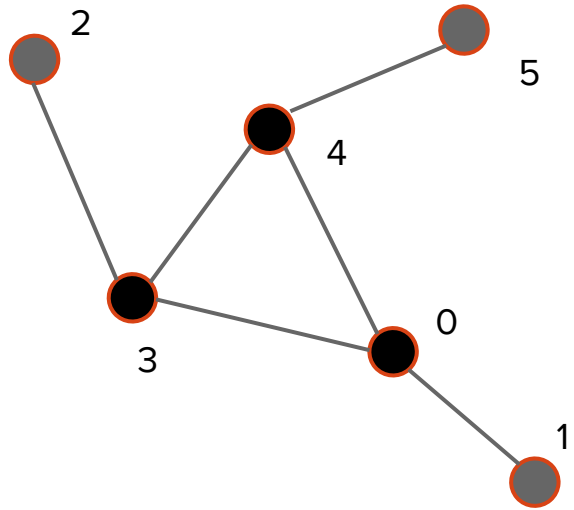
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

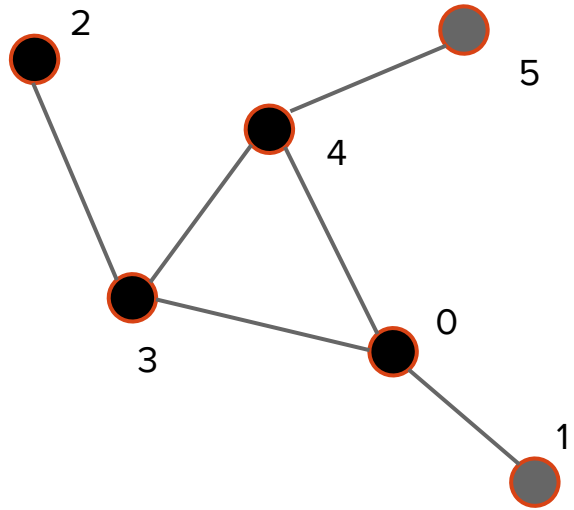
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

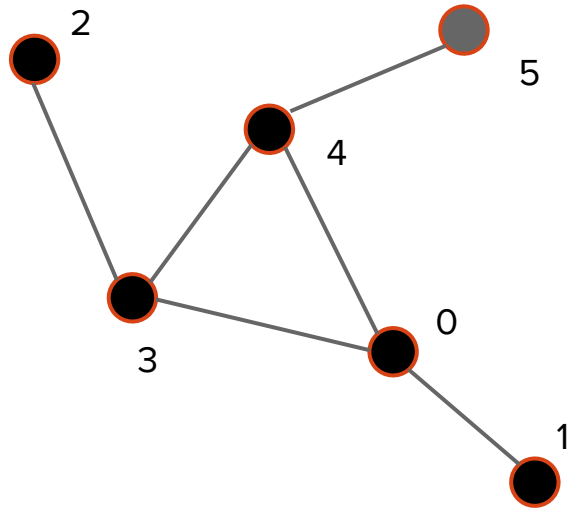
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

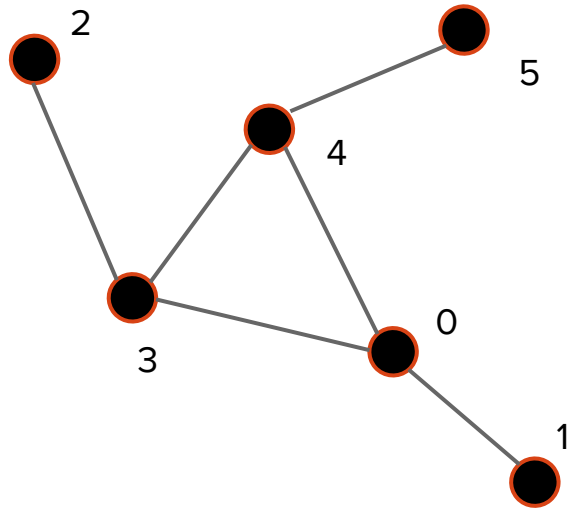
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Un recorrido por anchura (breadth-first search) comienza en un vértice inicial y visita a los vértices por distancia

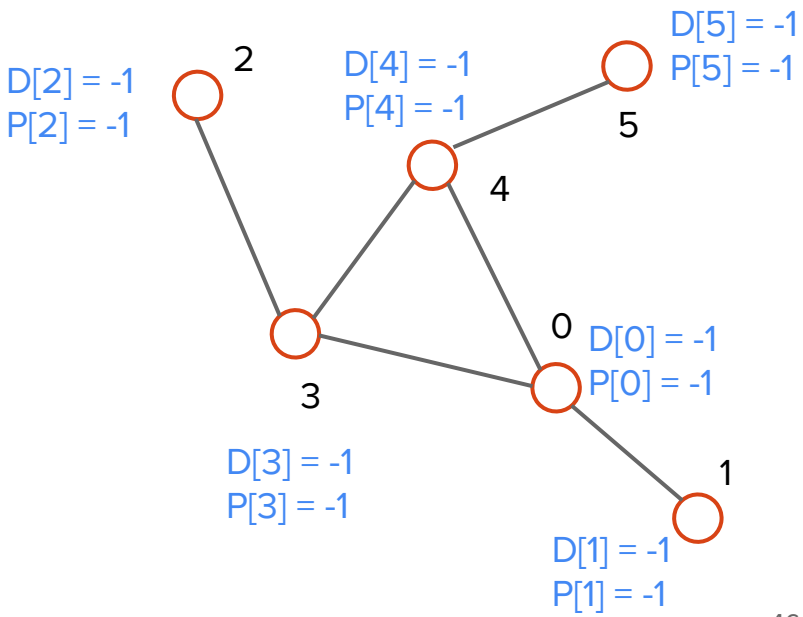
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      color[inicial] = GRIS;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            G[v] = NEGRO;
            for(auto &w: G[v])
                  if( color[w] == BLANCO ){
                        Q.push(w);
                        color[w] = GRIS;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

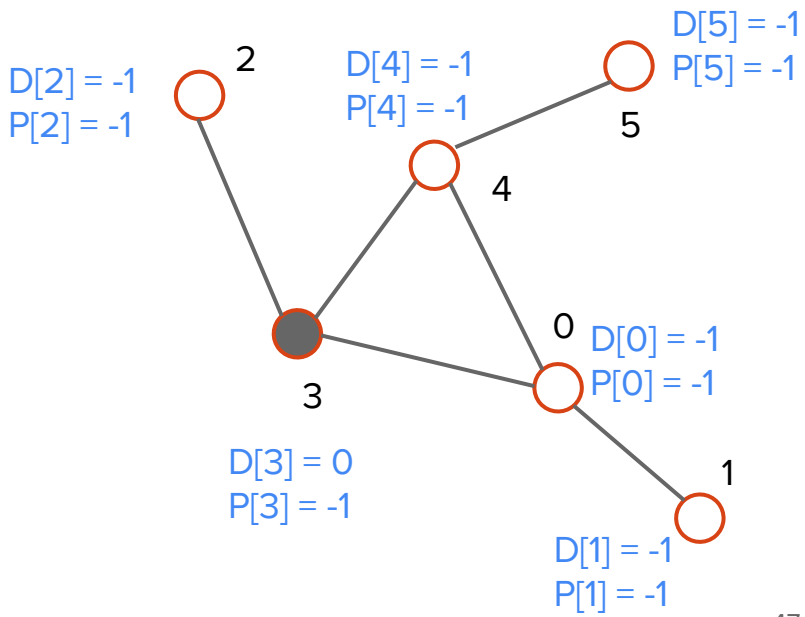
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

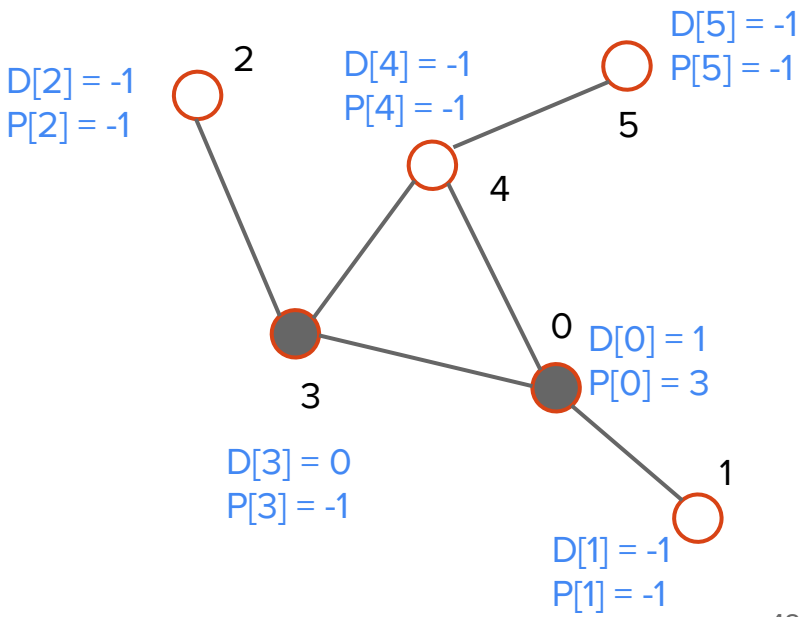
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

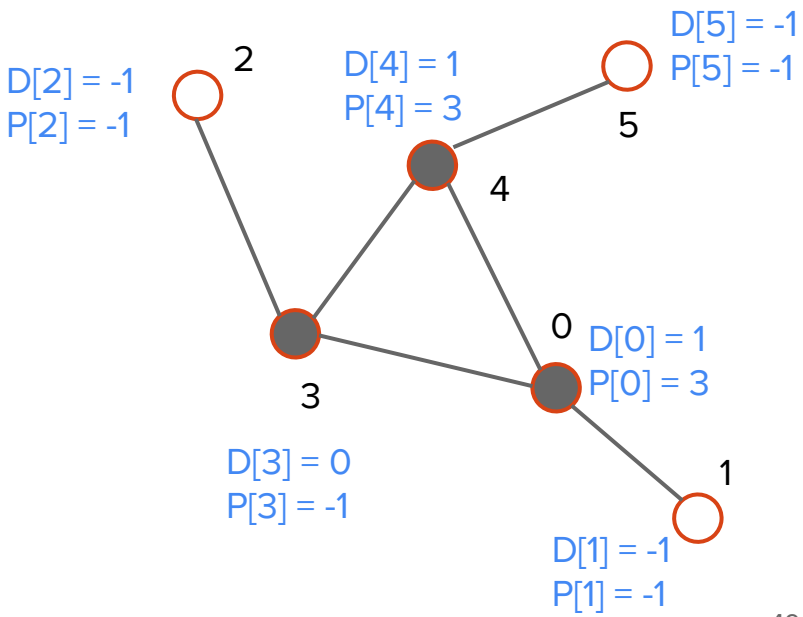
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

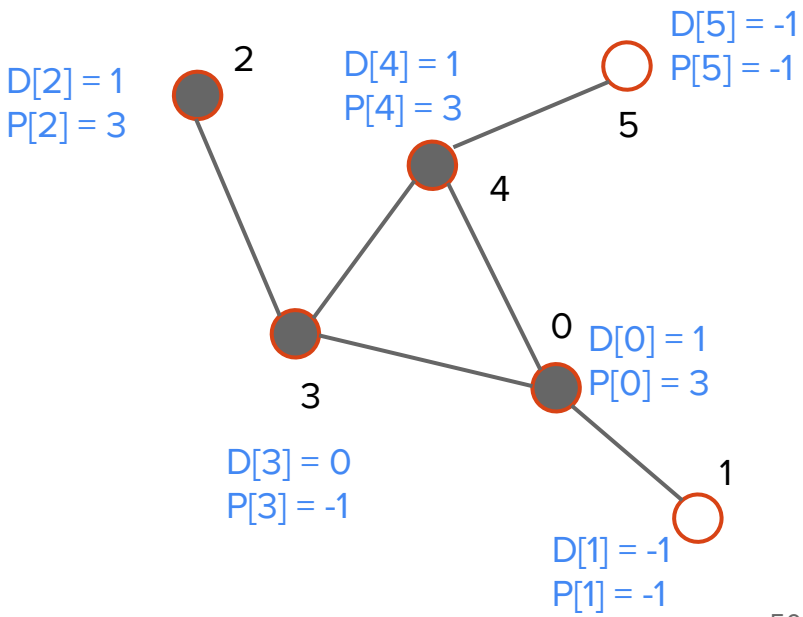
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

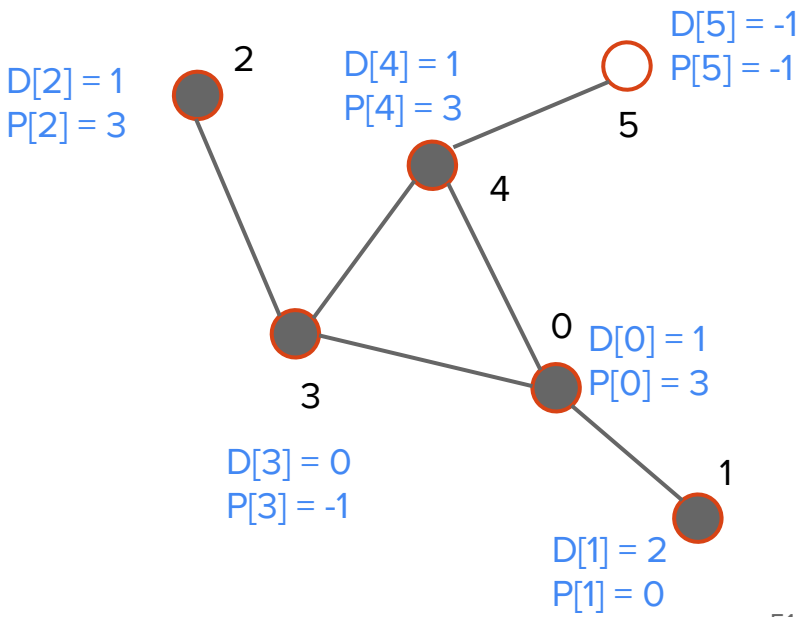
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

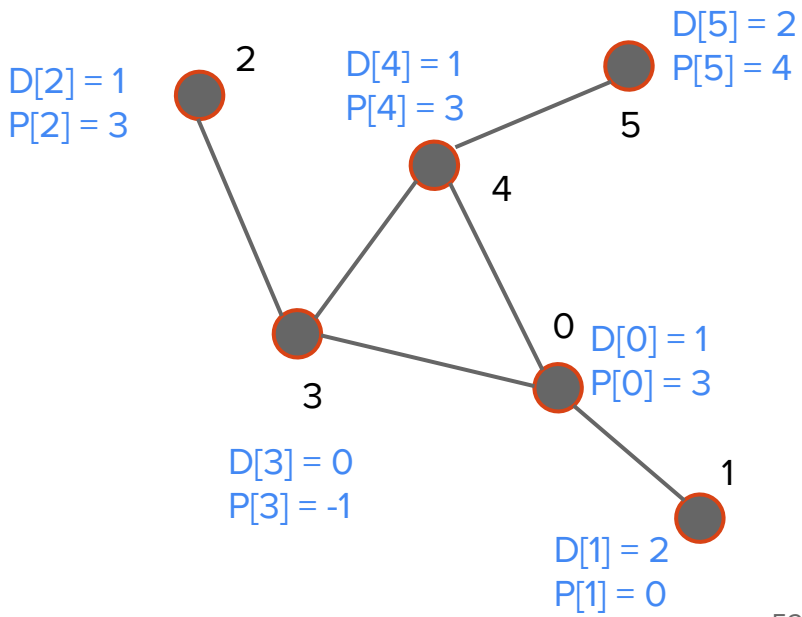
```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



BFS

Podemos calcular dichas distancias en un vector D, y los padres en P.

```
void bfs(int inicial){
      queue<int> Q;
      Q.push(inicial);
      D[inicial] = 0;
      while(not Q.empty()){
            int v = Q.front(); Q.pop();
            for(auto &w: G[v])
                  if( D[w] == -1 ){
                        Q.push(w);
                        D[w] = D[v] + 1;
                        P[w] = v;
                  }
      }
}
```



Análisis

Bajo la representación de lista de adyacencias, ambos algoritmos iteran cada arco exactamente una vez, y cada vértice es visitado una única vez.

De donde resulta que el tiempo es O(N + M).

Si el grafo es conexo, O(N+M) = O(M).

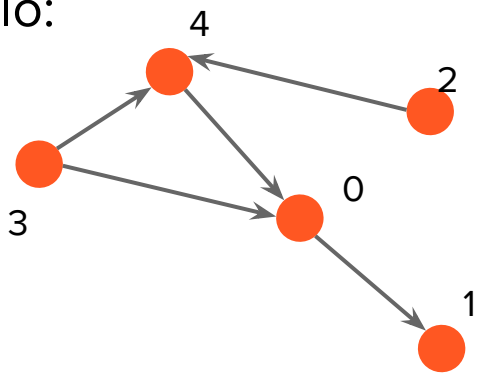
Aplicaciones

Orden topológico

Un grafo dirigido se denomina DAG si no tiene ciclos dirigidos.

Un ordenamiento topológico es un ordenamiento de los vértices de un grafo dirigido tal que todos los arcos quedan dibujados hacia la derecha.

Ejemplo:



posible toposort

Orden topológico

Teorema:

Un grafo es un DAG sii tiene un orden topológico.

¡ Podemos calcular uno con dfs !

Mantenemos una lista en espacio global y antes de retornar de visitar un vértice, lo anexamos al final de la lista.

Una vez visitados todos los vértices, revertimos la lista.

Orden topológico

```
vector<int> toposort;

void dfs(int v){
        visitado[v] = true;
        for(auto &w: G[v])
                if( not visitado[w] )
                        dfs(w);
        toposort.push_back(v);
}
...
forn(v, N)
        if( not visitado[v] )
                dfs(v);
reverse(toposort.begin(), toposort.end());
```

Más aplicaciones DFS

- Hallar componentes conexas
- Hallar componentes biconexas
  - Hallar puntos de articulación y puentes
- Hallar componentes fuertemente conexas
  - 2-SAT
- Test bipartición
- Test de planaridad
- Hallar ciclo Eulerianos

Kosaraju's

Es un algoritmo para hallar las componentes fuertemente conexas de un grafo dirigido.

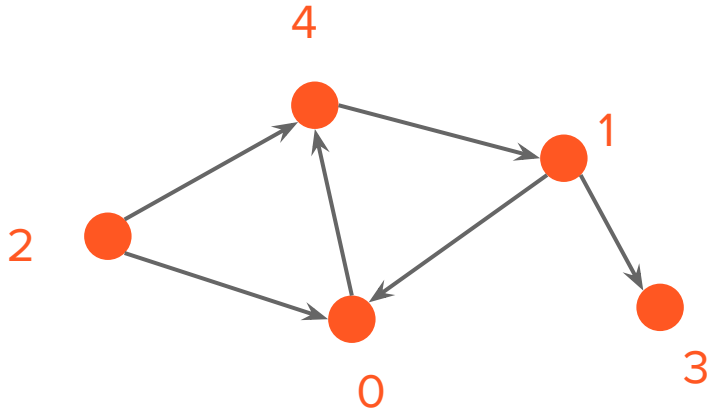
¿Qué pasaría si intentáramos obtener un toposort de un grafo que no sea un DAG?

Obtendríamos un ordenamiento que preserva el orden (topológico) de las SCCs, aunque los vértices de una misma SCC estén desordenados.

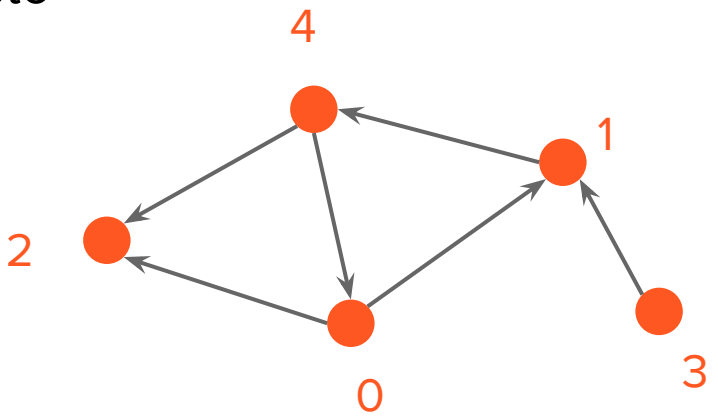
Kosaraju's

Supongamos que obtenemos:

2 4 1 0 3



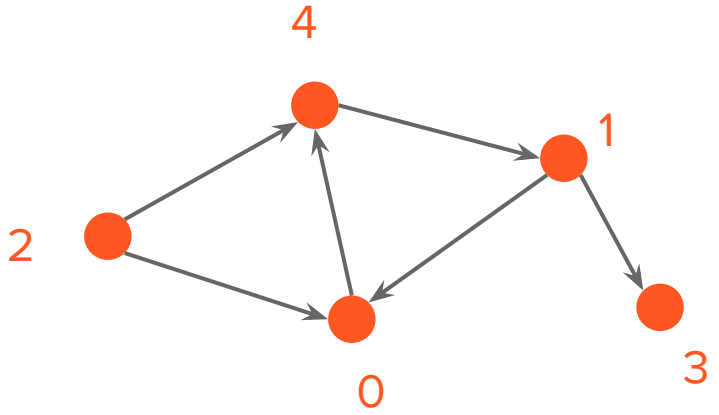
Si tiramos un DFS en el grafo transpuesto en este orden, obtenemos las SCCs



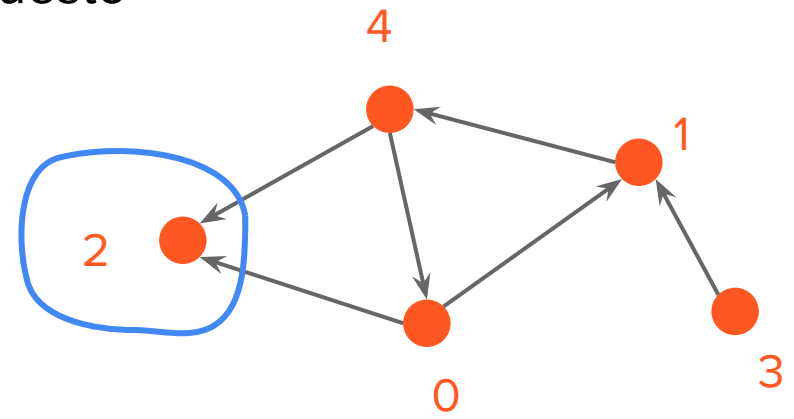
Kosaraju's

Supongamos que obtenemos:

2 4 1 0 3



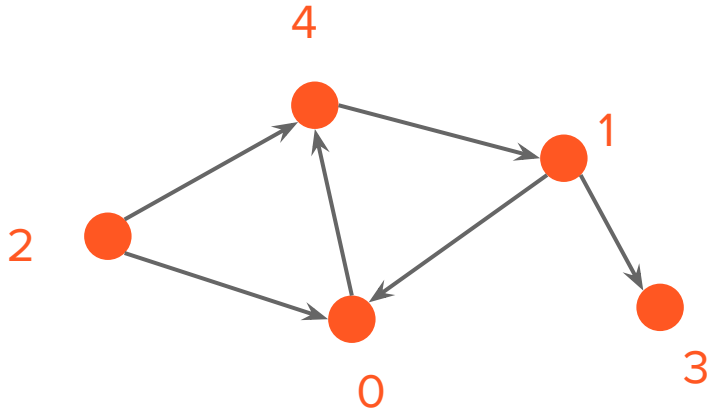
Si tiramos un DFS en el grafo transpuesto en este orden, obtenemos las SCCs



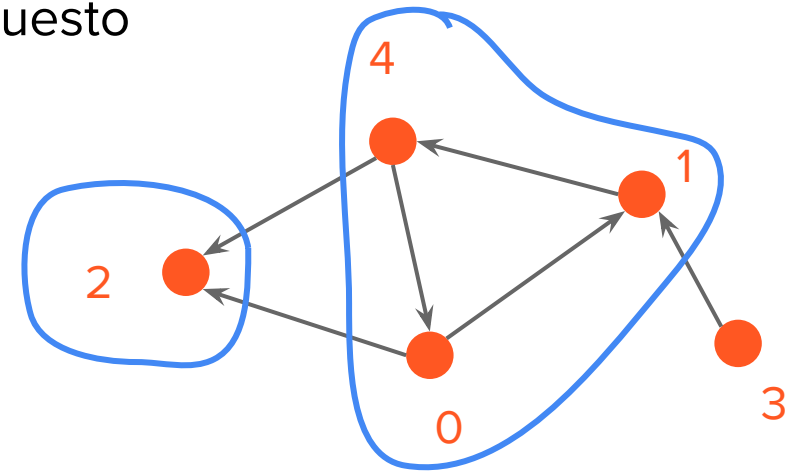
Kosaraju's

Supongamos que obtenemos:

2 4 1 0 3



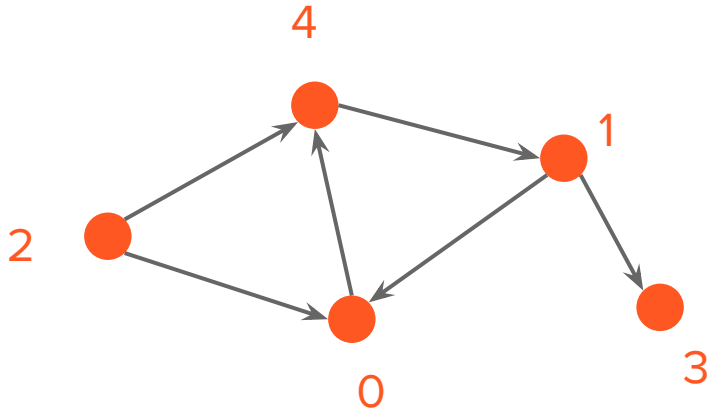
Si tiramos un DFS en el grafo transpuesto en este orden, obtenemos las SCCs



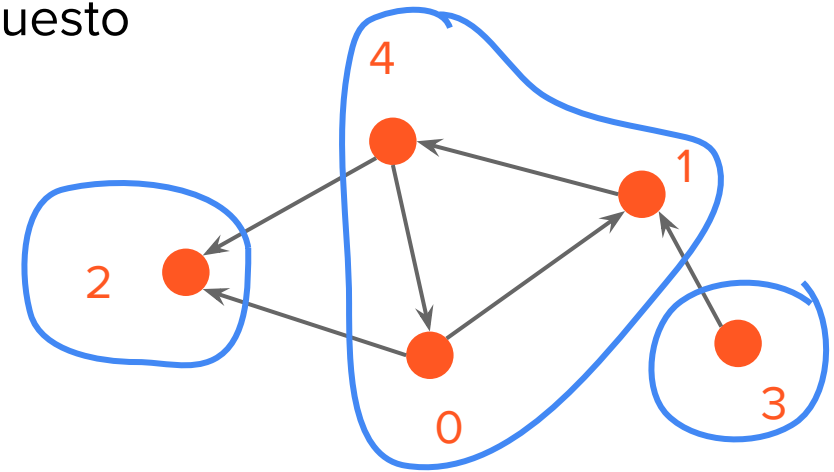
Kosaraju's

Supongamos que obtenemos:

2 4 1 0 3



Si tiramos un DFS en el grafo transpuesto en este orden, obtenemos las SCCs



2-SAT

Toda fórmula proposicional (*PROP*) puede obtenerse inductivamente:

- x es una variable $\Rightarrow$ x $\in$ *PROP*
- x $\in$ *PROP* $\Rightarrow$ ($\neg$x) $\in$ *PROP*
- x, y $\in$ *PROP* $\Rightarrow$ (x $\bigstar$ y) $\in$ *PROP* donde $\bigstar \in \{ \wedge, \vee, \rightarrow \}$

Una fórmula está en k-CNF si puede escribirse como una conjunción de disjunciones de hasta k variables (negadas o no).

Queremos ver si una fórmula es satisfacible (es decir, es posible encontrar una valuación de variables tales que la fórmula da verdadero)

2-SAT

Infortunadamente k-SAT es NP-Completo para k > 2.

1-SAT es trivial
2-SAT es interesante!

Podemos construir el grafo de implicancias de la fórmula:

- Los vértices son las variables y sus negaciones.
- Hay un arco de u a v si u implica v cuando vale la fórmula.

La fórmula es satisfacible sii una variable y su negación no están en la misma SCC.

2-SAT

Ejemplo:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

Resulta en las siguientes implicancias:

$\neg a \rightarrow \neg b$

$b \rightarrow a$

$a \rightarrow b$

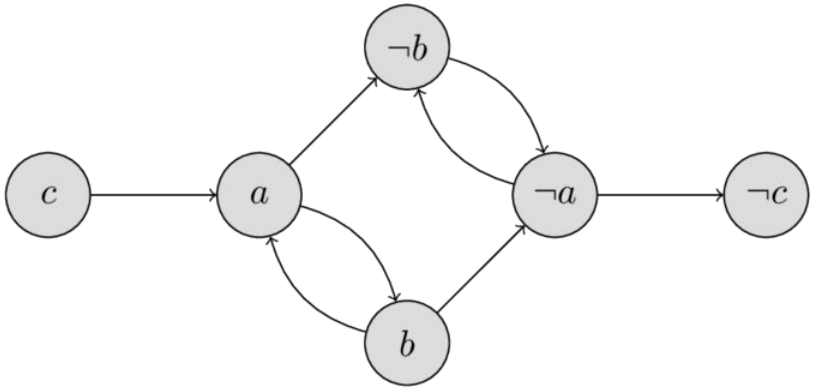
$\neg b \rightarrow \neg a$

$a \rightarrow \neg b$

$b \rightarrow \neg a$

$\neg a \rightarrow \neg c$

$c \rightarrow a$



ejemplo extraído de http://e-maxx.ru

2-SAT

Ejemplo:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

Resulta en las siguientes implicancias:

$\neg a \rightarrow \neg b$

$b \rightarrow a$

$a \rightarrow b$

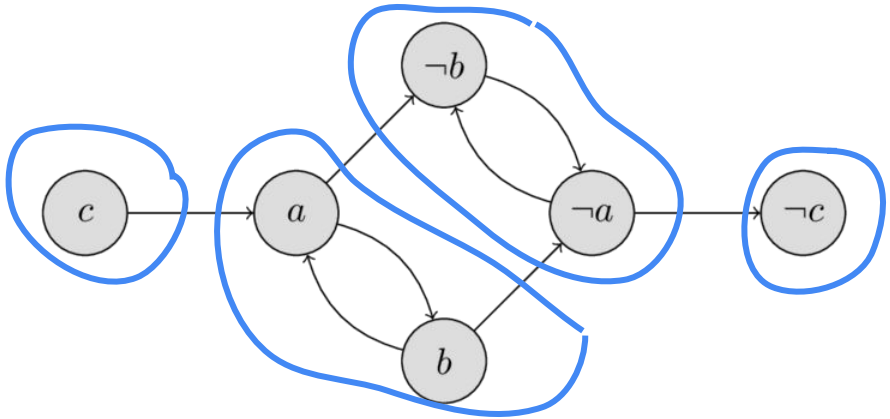
$\neg b \rightarrow \neg a$

$a \rightarrow \neg b$

$b \rightarrow \neg a$

$\neg a \rightarrow \neg c$

$c \rightarrow a$



ejemplo extraído de http://e-maxx.ru

Puentes

Existe un algoritmo por el Dr. Robert Tarjan para encontrar componentes biconexas, puntos de articulación (vértices de corte) y puentes de un grafo en O(N) con un solo DFS.

Sin embargo, siguiendo la onda de SCC, podemos encontrar los **puentes** mediante el siguiente algoritmo:

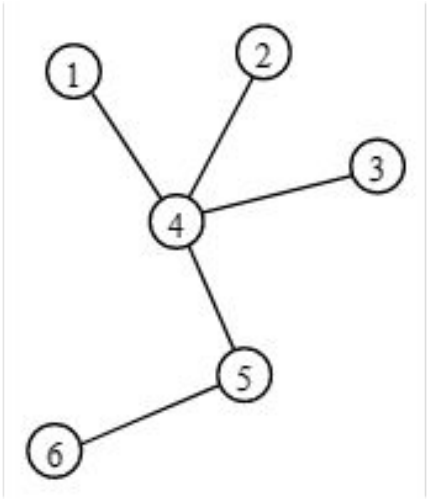
- Tirar un DFS, orientar las aristas en el sentido en las cual las recorremos
- Hallar las SCCs del grafo
- Una arista u-v es un puente sii u y v pertenecen a SCCs distintas

Árbol Generador Mínimo (MST)

esto no es un MST

MST

Un árbol (no dirigido) es un grafo tal que:

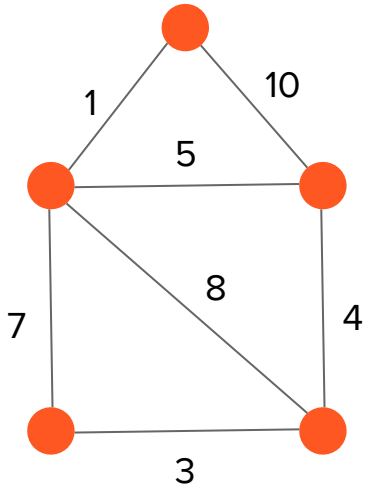


- es conexo
- es acíclico

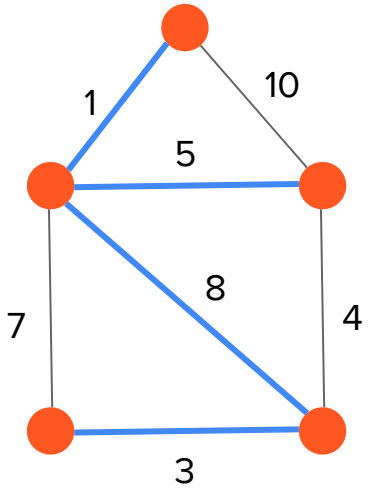
Equivalentemente, podemos decir:

- existe un único camino entre cada par de vértices

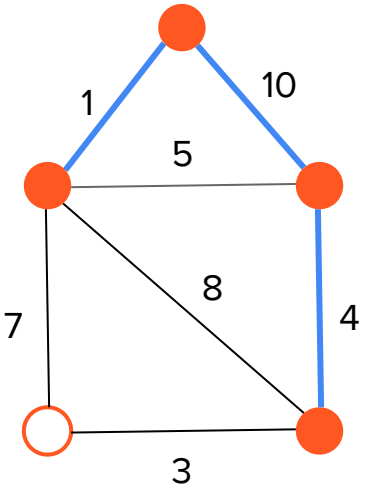
MST



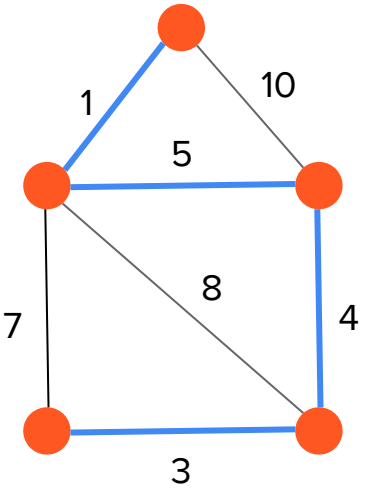
grafo original



árbol generador
peso = 17



árbol no generador
peso = 15



MST
peso = 13

Kruskal

Kruskal

El algoritmo de Kruskal halla un MST de la siguiente manera:

- ordenamos las aristas por peso creciente
- MST = vacío
- por cada arista e:
    si MST + e no forma un ciclo:
        MST += e

La única cosa difícil de este algoritmo es verificar si formamos un ciclo o no. Esto se resuelve con la estructura de conjuntos disjuntos (AKA union-find)

Union Find

La estructura de conjuntos disjuntos tiene 3 operaciones:

- init(n) crea n conjuntos disjuntos
- find(x) devuelve el ID del conjunto donde está x
- join(x, y) une los conjuntos donde están x e y

find y join se realizan muy eficientemente, O(1) a fines prácticos.

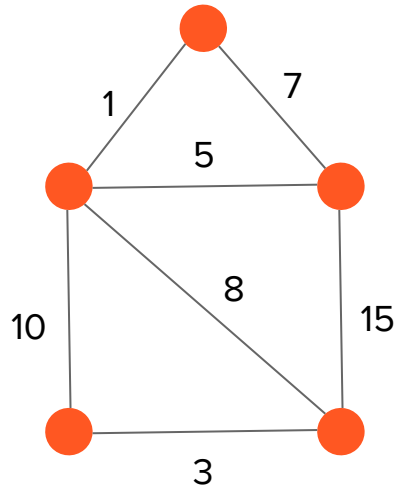
¡Y su implementación es muy simple!

Union Find

```
struct disjoint_sets{
    void init(int n){
        v.clear();
        v.resize(n, -1);
    }
    int find(int x){
        return v[x] == -1 ? x : v[x] = find(v[x]);
    }
    int join(int x, int y){
        x = find(x); y = find(y);
        if( x != y )
            v[x] = y;
    }
    vector<int> v;
};
```

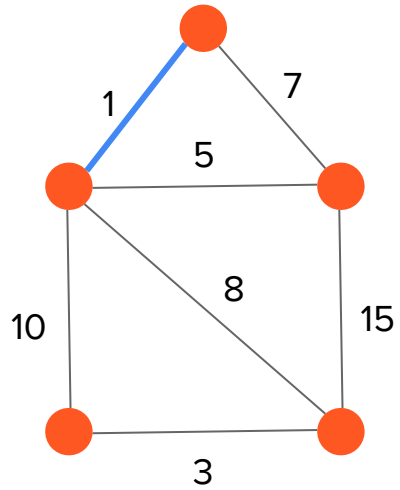
Kruskal

```
// Sea E la lista de aristas {u, v, peso} de un grafo conexo
// Sea S la estructura de conjuntos disjuntos

sort(E.begin(), E.end());
for(auto &e: E)
     if( S.find(e.u) != S.find(e.v) )
          Ans += e.peso;
     else S.join(e.u, e.v);

cout << "Peso del MST: " << Ans << endl;
```



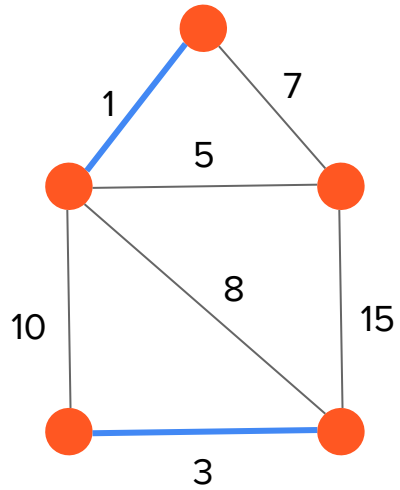
Kruskal

```
// Sea E la lista de aristas {u, v, peso} de un grafo conexo
// Sea S la estructura de conjuntos disjuntos

sort(E.begin(), E.end());
for(auto &e: E)
     if( S.find(e.u) != S.find(e.v) )
          Ans += e.peso;
     else S.join(e.u, e.v);

cout << "Peso del MST: " << Ans << endl;
```



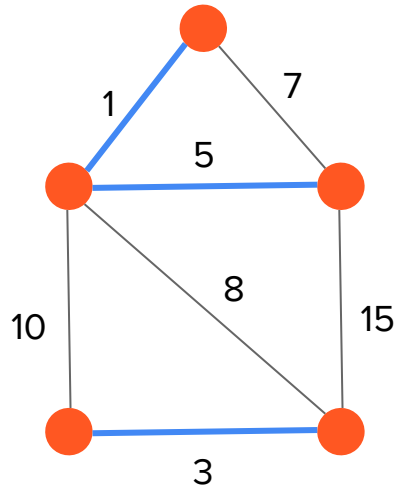
Kruskal

```
// Sea E la lista de aristas {u, v, peso} de un grafo conexo
// Sea S la estructura de conjuntos disjuntos

sort(E.begin(), E.end());
for(auto &e: E)
     if( S.find(e.u) != S.find(e.v) )
          Ans += e.peso;
     else S.join(e.u, e.v);

cout << "Peso del MST: " << Ans << endl;
```



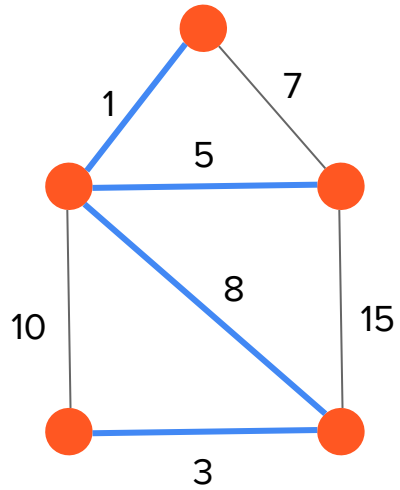
Kruskal

```
// Sea E la lista de aristas {u, v, peso} de un grafo conexo
// Sea S la estructura de conjuntos disjuntos

sort(E.begin(), E.end());
for(auto &e: E)
     if( S.find(e.u) != S.find(e.v) )
          Ans += e.peso;
     else S.join(e.u, e.v);

cout << "Peso del MST: " << Ans << endl;
```



Kruskal

```
// Sea E la lista de aristas {u, v, peso} de un grafo conexo
// Sea S la estructura de conjuntos disjuntos

sort(E.begin(), E.end());
for(auto &e: E)
     if( S.find(e.u) != S.find(e.v) )
          Ans += e.peso;
     else S.join(e.u, e.v);

cout << "Peso del MST: " << Ans << endl;
```



Análisis

Ordenar las aristas por peso creciente es O(M log M).

Luego, por cada iteración hacemos a lo sumo dos finds y un join.
Ambas operaciones son prácticamente O(1).
Como son M iteraciones, en total quedaría O(M).

Por lo que el algoritmo completo es O(M log M) por el ordenamiento.
Sin embargo, si las aristas ya vienen ordenadas el algoritmo es O(M).

Distancias en grafos ponderados

Dijkstra

Dijkstra

Cuando tenemos grafos ponderados, el BFS ya no nos sirve para calcular la distancia más corta desde un vértice a los demás.

Por suerte existe un algoritmo que resuelve el problema: Dijkstra.

El algoritmo es similar al BFS, iremos metiendo los caminos descubiertos en una bolsa de prioridad, y en cada iteración extraeremos el de menor longitud e intentaremos extenderlo.

Dijkstra

```
struct hedge{
      int v, d;
      bool operator<(const arco &otro) const {
            return d > otro.d;
      }
};

vector<hedge> G[MAXN];
```

La estructura hedge guarda "medio arco": la longitud y el vértice final.

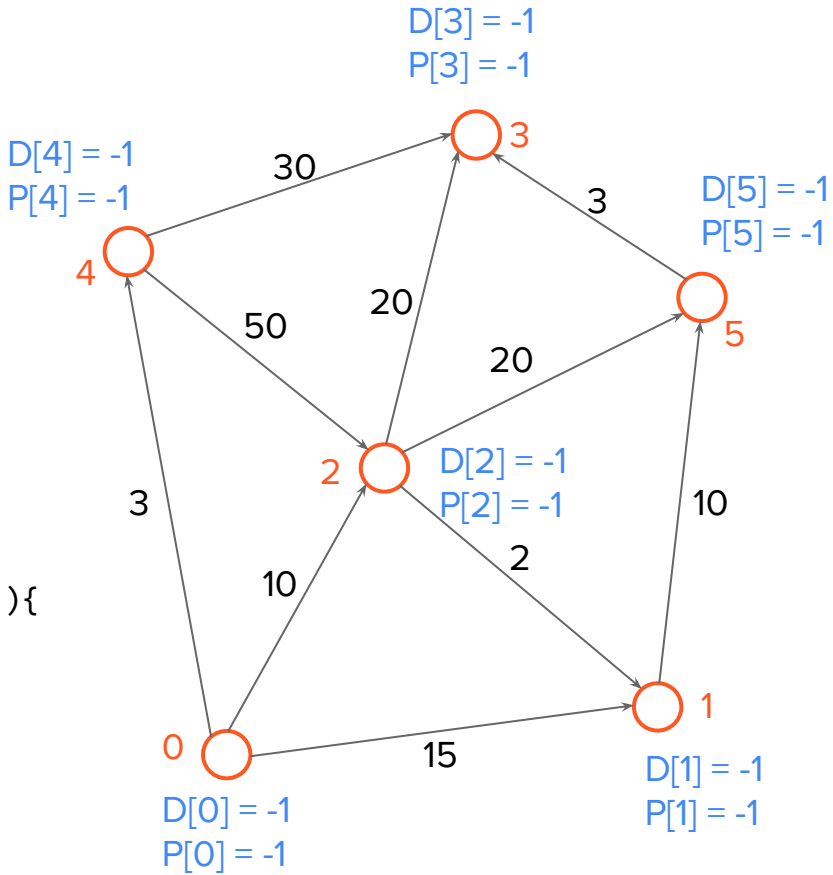
Para insertar el arco u➔v con peso d:
`G[u].push_back({v, d});`

Reutilizaremos la misma estructura para representar el camino de longitud d que termina en v.

Diremos que un camino tiene **menos prioridad** que otro si su **longitud es mayor**.

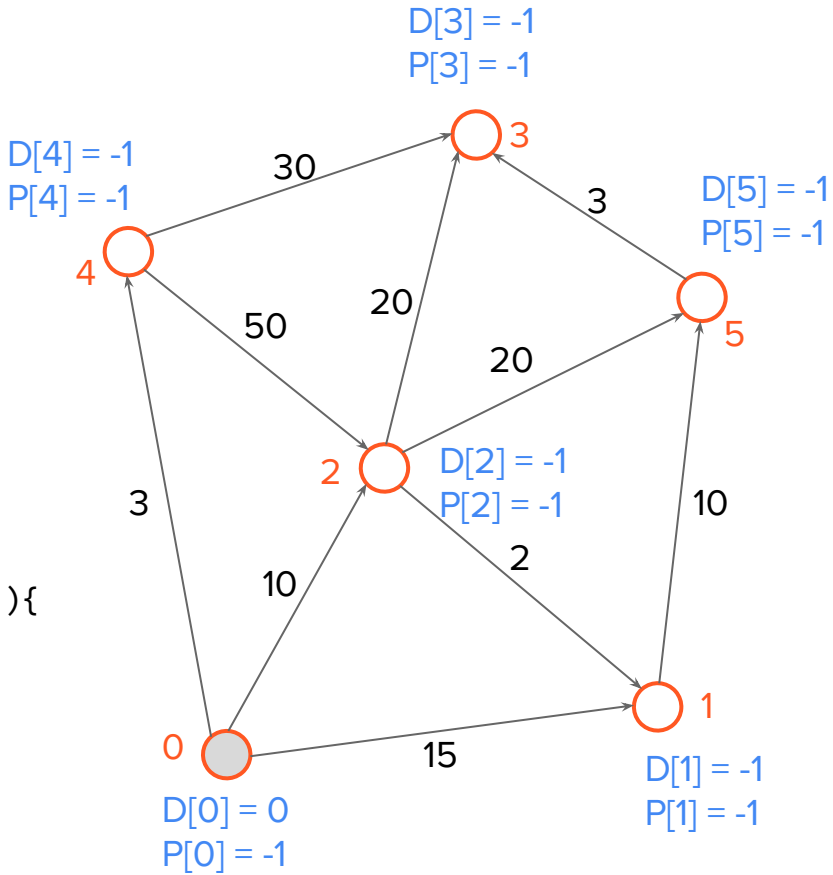
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



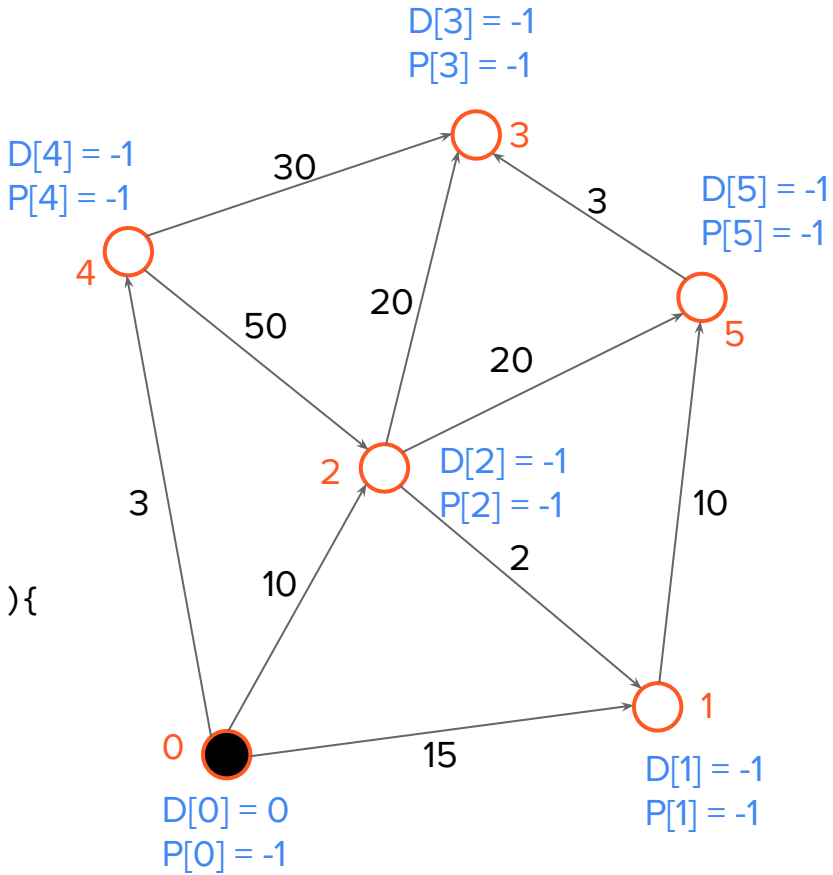
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



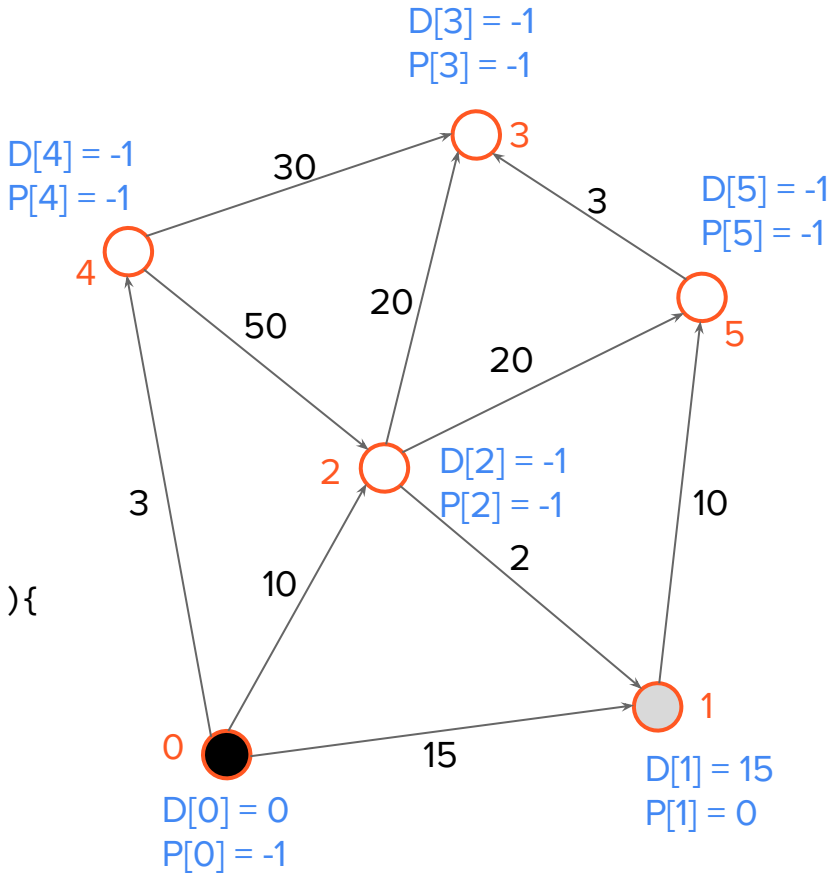
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



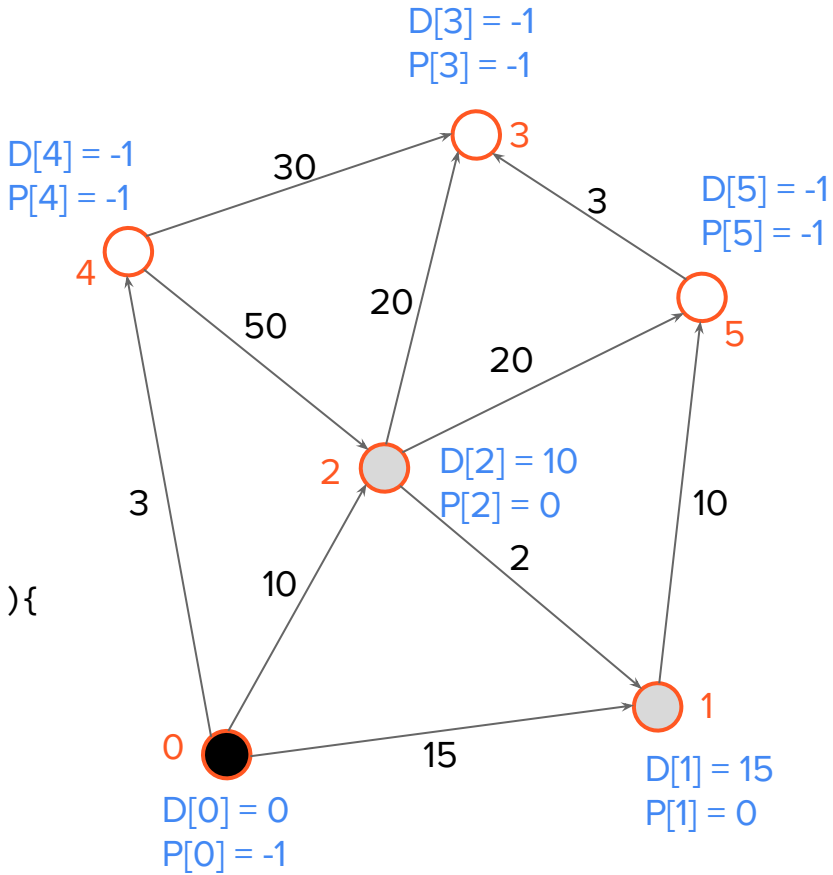
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



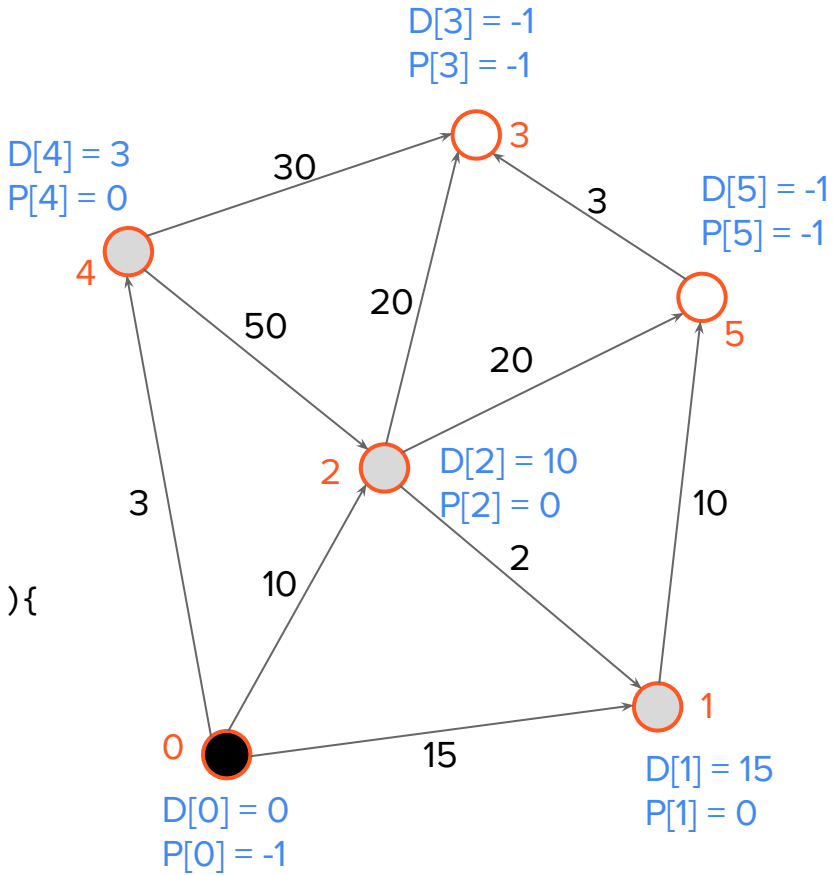
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



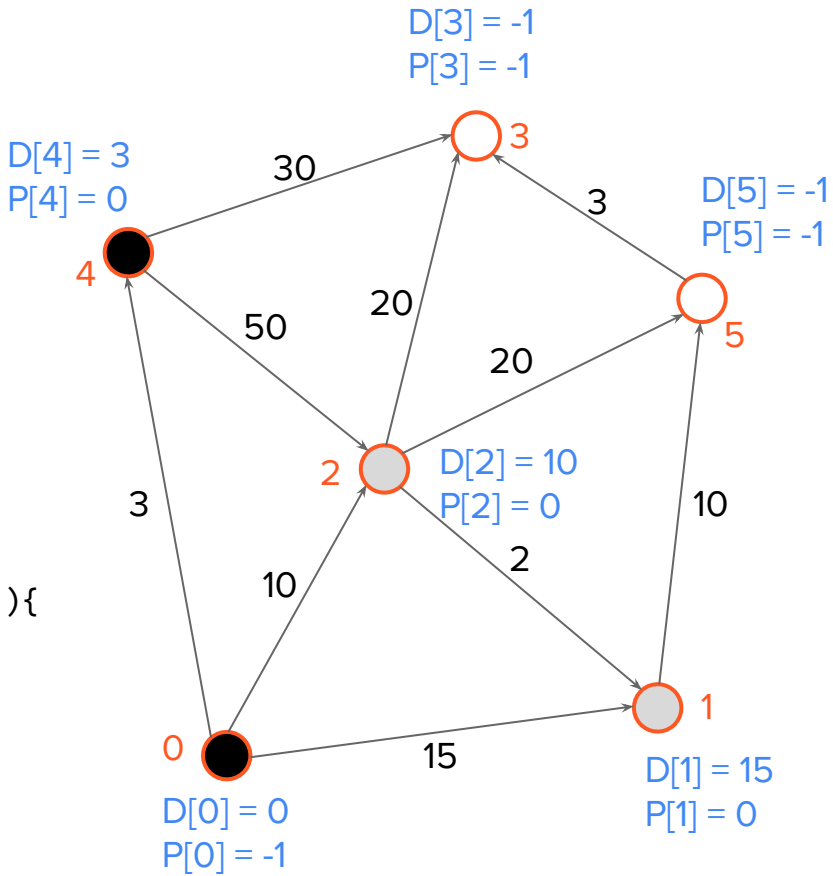
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



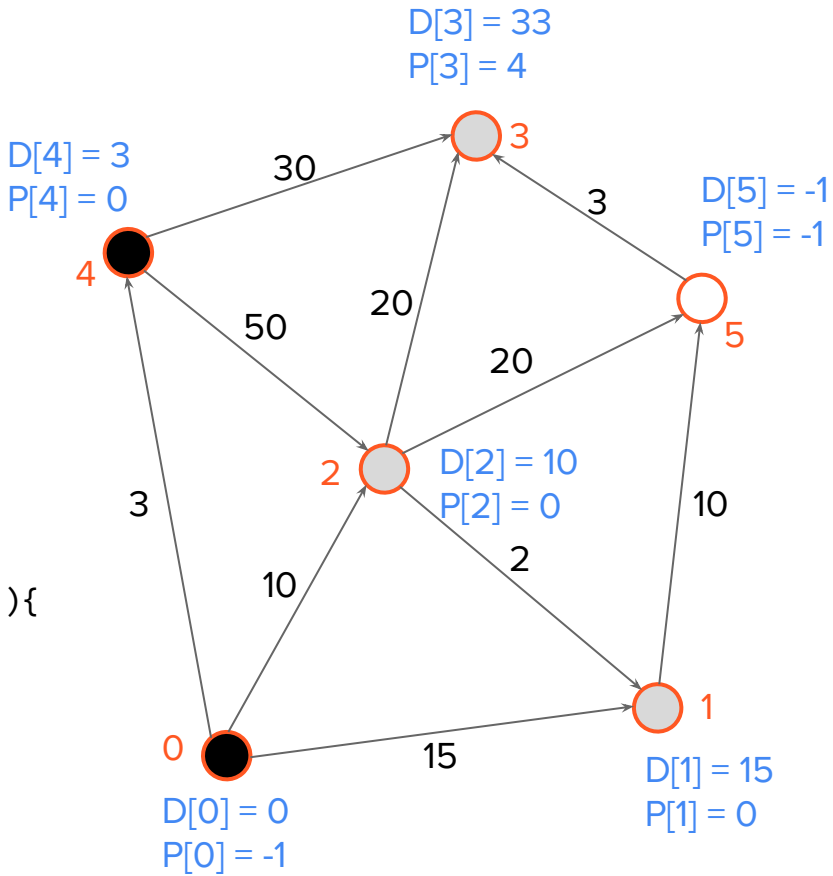
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



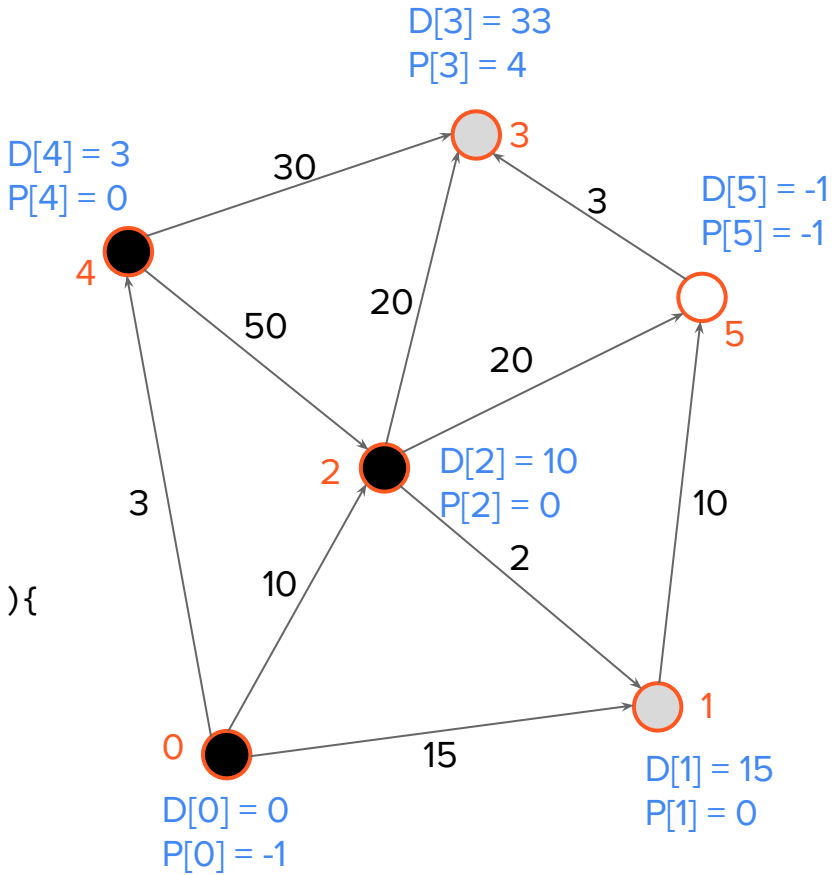
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



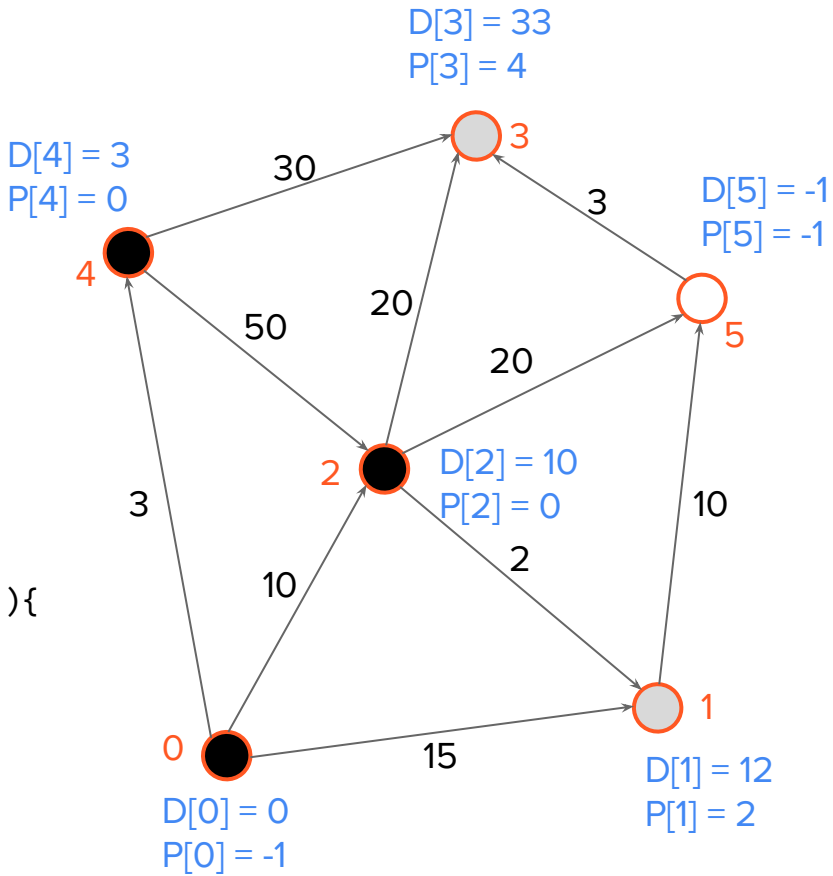
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



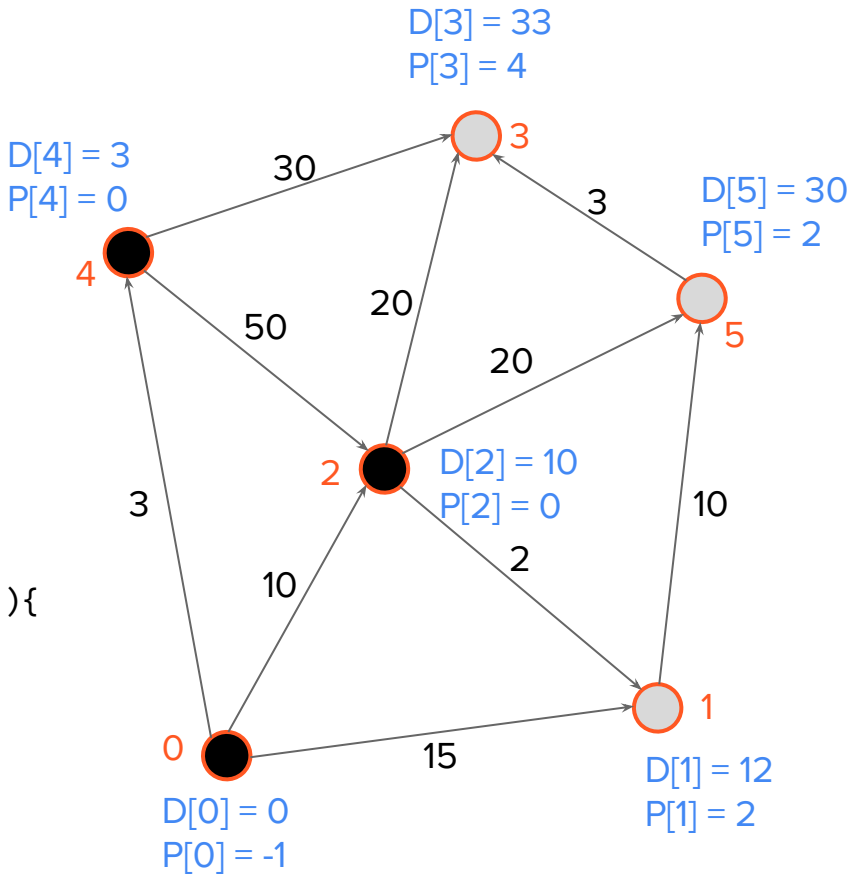
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



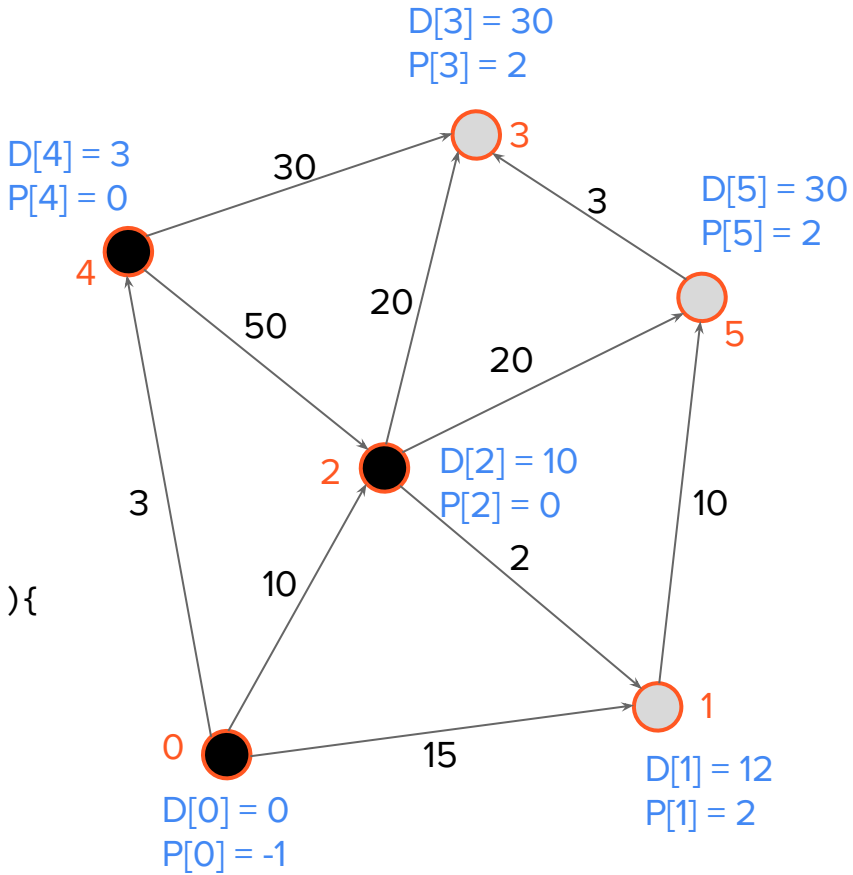
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



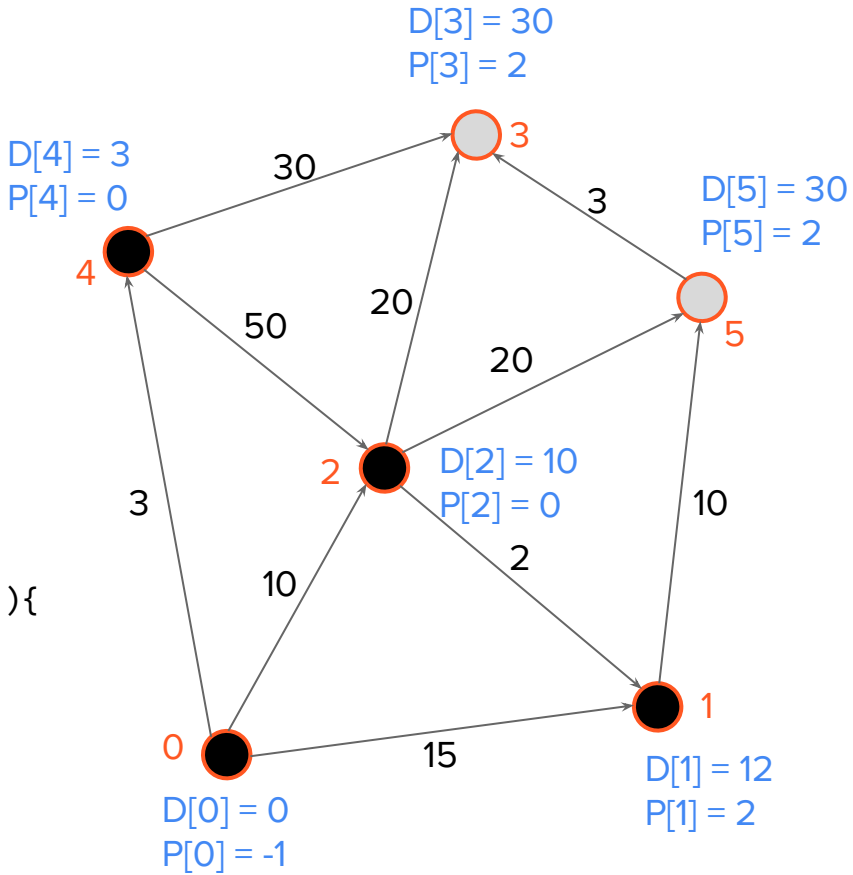
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



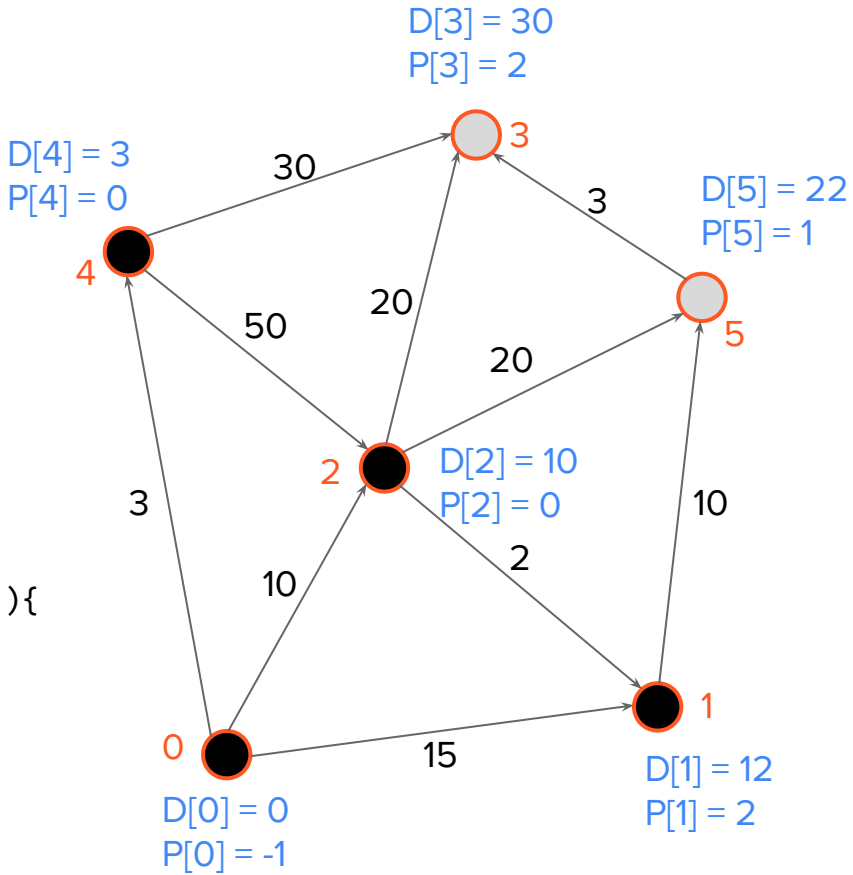
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



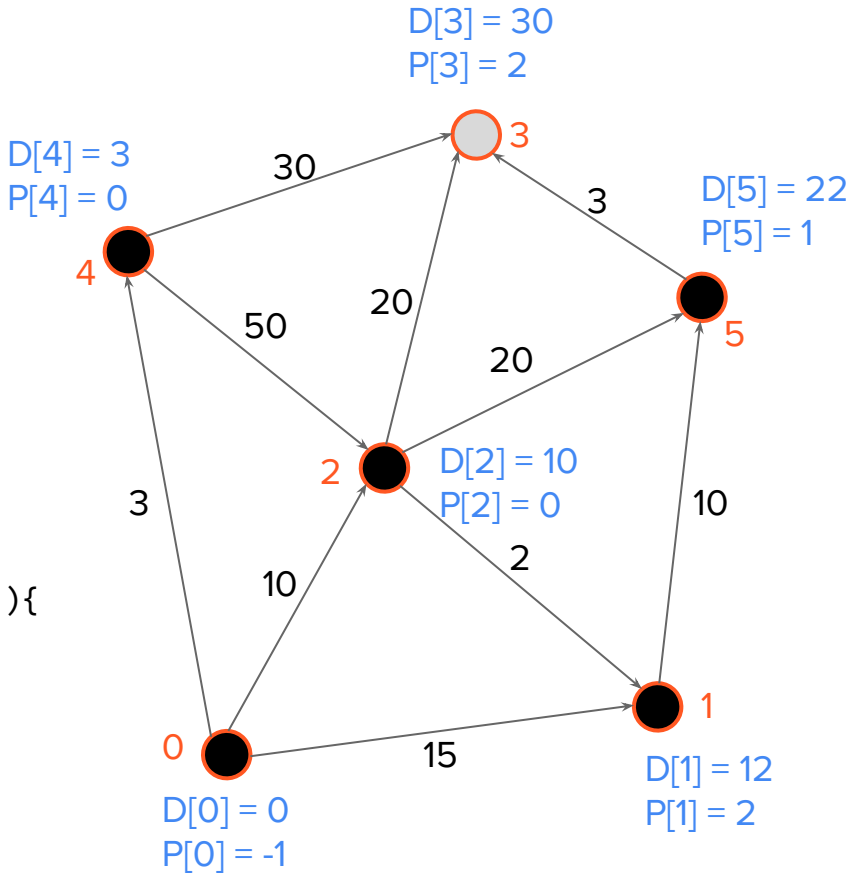
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



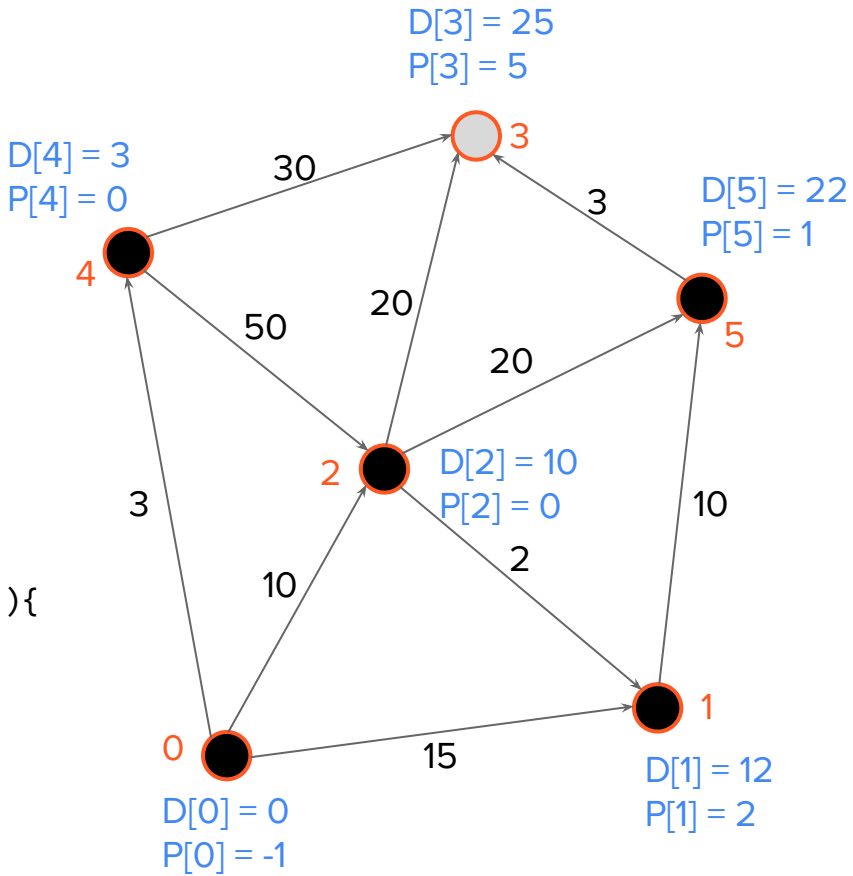
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



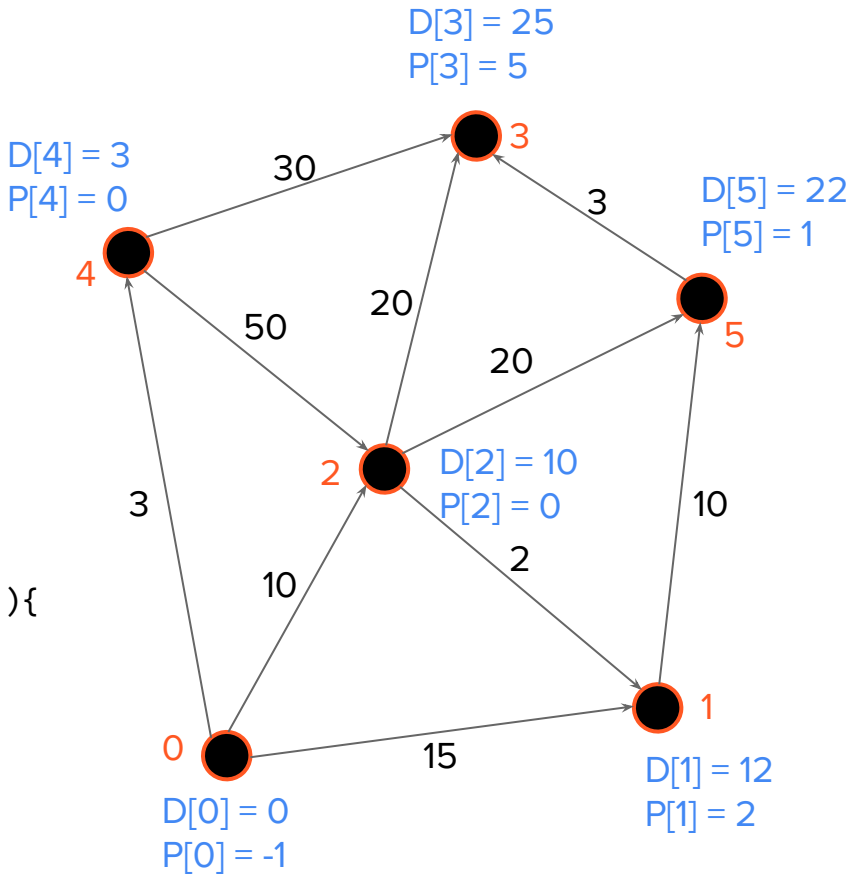
Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



Dijkstra

```
void Dijkstra(int inicial){
      priority_queue<hedge> Q;
      Q.push({inicial, 0});
      D[inicial] = 0;
      while(not Q.empty()){
            auto c = Q.top(); Q.pop();
            if( D[c.v] < c.d )
                  continue;
            for(auto &e: G[c.v])
                  if( D[e.v] == -1 or D[e.v] > c.d + e.d ){
                        D[e.v] = c.d + e.d;
                        P[e.v] = c.v;
                        Q.push({e.v, D[e.v]});
                  }
      }
}
```



Análisis

Bajo la representación de lista de adyacencias, y usando una binary heap como priority queue, la complejidad temporal es O(M log N).

Floyd Warshall

Floyd-Warshall

Si quisiéramos calcular el camino más corto entre cualquier par de vértices, podríamos...

Si el grafo no es ponderado, tirar un BFS desde cada vértice en tiempo O(NM).

Si el grafo es ponderado, tirar un Dijkstra desde cada vértice en tiempo O(NM log N).

Floyd-Warshall

Si el grafo es muy denso ($M \sim N^2$), muchos Dijkstras van a dar una complejidad peor que N^3.

Por suerte existe un algoritmo que:

- Corre en $O(N^3)$
- Es trivial de programar
- No se rompe con arcos con pesos negativos

Floyd-Warshall

La idea es calcular mediante programación dinámica la siguiente tabla:

`D[i][j][k]` :=
longitud de un i-j camino mínimo cuyos vértices intermedios están en 1..k

Vemos que `D[i][j][k]` es el mínimo de:

`D[i][j][k-1]`	← el camino mínimo no pasa por k
`D[i][k][k-1] + D[k][j][k-1]`	← el camino mínimo pasa por k

Floyd-Warshall

Guardaremos en una matriz D de NxN la distancia más corta para todo par de vértices.
INF representa un número muy grande.

```
forn(i, N) forn(j, N) D[i][j] = i == j ? 0 : INF;

for(auto &e: E) D[e.u][e.v] = min(D[e.u][e.v], e.d);


forn(k, N)
    forn(i, N)
        forn(j, N)
            D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
```

Floyd-Warshall

Podemos rápidamente reconstruir los caminos guardando el siguiente vértice:

```
for(auto &e: E){
     D[e.u][e.v] = e.d;
     Next[e.u][e.v] = e.v;
}

forn(k, N)
     forn(i, N)
          forn(j, N)
               if( D[i][j] > D[i][k] + D[k][j] ){
                    D[i][j] = D[i][k] + D[k][j];
                    Next[i][j] = Next[i][k];
               }
```

Gracias!