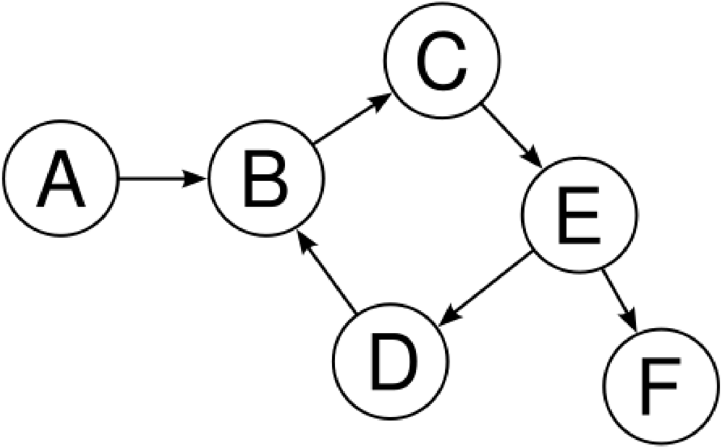
Grafos dirigidos

May 8, 2018

Qué es un grafo dirigido?

- Es un conjunto de vértices que están conectados de alguna manera entre sí (o no) mediante aristas.
- Es casi lo mismo que un grafo no dirigido, pero con una gran diferencia: si existe un lado del vértice X al vértice Y, no necesariamente existe uno desde Y a X.

Ejemplo - grafo dirigido



Código para crear un grafo dirigido

- El grafo tiene N vértices y M lados
- Un lado se representa con una línea de input conteniendo dos enteros
- Se suele usar listas de adyacencia para representar grafos dirigidos

```
vector <int> graph[MAX_N+1];

void build_graph(){
  int n, m;
  cin >> n >> m;
  for(int i = 0; i < m; i++){
    cin >> x >> y;
    graph[x].push_back(y)
  }
}
```

Ciclos en grafos dirigidos

- En la clase anterior vimos como encontrar ciclos en grafos no dirigidos.
- Funciona el mismo método para grafos dirigidos?

Ciclos en grafos dirigidos

- En la clase anterior vimos como encontrar ciclos en grafos no dirigidos.
- Funciona el mismo método para grafos dirigidos?
- No. Porque asume que si desde un vertice X puede llegar a uno que ya fue visitado, es porque lo visite en el mismo camino por el que llegué a X.
- Pero esto no siempre es verdad en un grafo dirigido.

Ciclos en grafos dirigidos

- Necesitamos una forma de saber si el vertice ya visitado pertenece a un camino que no ha sido recorrido completamente todavía.
- Ideas?

Ciclos en grafos dirigidos

- Necesitamos una forma de saber si el vertice ya visitado pertenece a un camino que no ha sido recorrido completamente todavía.
- Ideas?
- Correr DFS en el grafo genera un arbol (donde un vertice tiene de padre a quien lo llamó en la recursión)
- Hay un ciclo si un vértice X intenta visitar a un vertice Y que ya fue visitado, y se cumple que Y es ancestro de X en el arbol generado por el DFS.
- En otras palabras, hay un ciclo en un grafo si un vertice puede ir a otro que ya fue visitado, y que todavía se encuentra en el stack de la recursión.

Código para detectar ciclos en grafos dirigidos

```
bool check(int v){
  visited[v] = true; recursion[v] = true;
  for(int u : graph[v]){
    if(!visited[u] && check(u))
      return true;
    else if(recursion[u])
      return true;
  }
  recursion[v] = false;
  return false;
}

bool check_cycles(int n){
  for(int i = 0; i < n; i++){
    if(!visited[i] && check(i)){
    return true;
    }
  }
  return false;
}
```

Topological sort

- Es un algoritmo sobre grafos acíclicos que devuelve un ordenamiento de sus vértices.
- Si hay un lado que conecta el vértice X con el vértice Y, X viene antes que Y en el orden topológico.
- Consiste en correr DFS en el grafo desde un nodo X arbitrario, y luego de llamar a la recursión sobre cada uno de los vecinos, agregar X al final del vector con el orden resultado.
- Notar que como X se agrega **luego** de la recursion sobre los vecinos, en el vector resultado, va estar después que todos los vértices a los que se podía llegar desde él.
- Justamente por esto, la etapa final del algoritmo es revertir el orden del arreglo resultado. De esta manera, se cumple la condición del algoritmo.

Código de topological sort

```
void dfs(int v){
  visited[v] = true;
  for(int u : graph[v]){
    if(!visited[u]){
      dfs(u);
    }
  }
  res.push_back(v);
}

void topological_sort(int n){
  for(int i = 0; i < n; i++){
    if(!visited[i]){
      dfs(i);
    }
  }
  reverse(res.begin(), res.end());
}
```

Problema de práctica

- En este problema pueden evaluar sus conocimientos sobre ciclos en grafos dirigidos y topological sort:

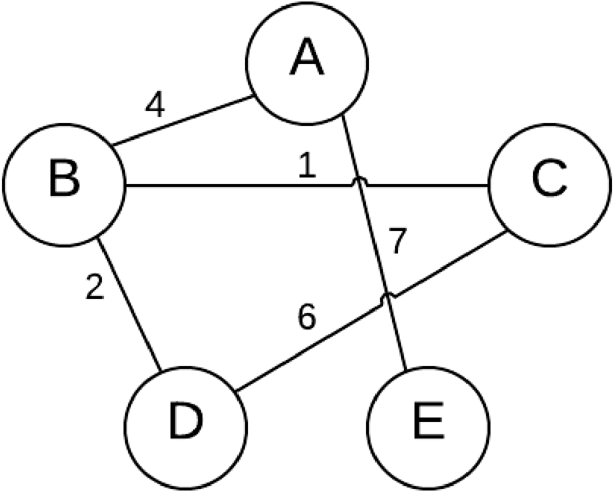
  `http://www.spoj.com/problems/TOPOSORT/`

- Ojo con orden topológico que imprimen (puede haber muchos válidos).

Grafos con pesos

- Hasta ahora vimos grafos donde los lados son simplemente conexiones entre dos vértices.
- Agregaremos un nuevo concepto: lados con pesos.
- Un lado con peso tiene, además de los dos vértices que conecta, un valor (conocido como peso).

Ejemplo - Grafo con pesos



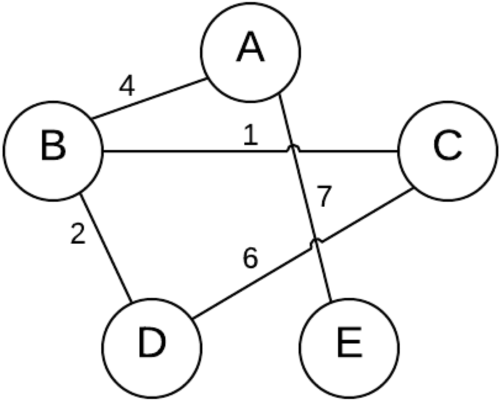
Caminos en grafos con pesos

- La definición de caminos en grafos con pesos es igual a la de grafos sin pesos.
- Peeeeero, se introduce el concepto de **costo** de un camino.
- El **costo de un camino** que parte desde el vértice X hasta el vértice Y se define como la suma de los pesos de los lados involucrados en ese recorrido.
- Los lados en un grafo sin pesos se pueden ver como lados con peso igual a 1.

Costo de un camino

Volviendo al grafo anterior:

- Cuáles son los posibles costos de ir desde A hasta E?
- Cuáles son los posibles costos de ir desde D hasta B?



Camino mínimo

- El **camino mínimo** desde un vertice X hasta un vértice Y es el mínimo costo entre todos los caminos posibles que parten en X y terminan en Y.
- Hay varios algoritmos que se encargan de computar caminos mínimos. En esta clase veremos uno de los más famosos: el algoritmo de Dijkstra.

Algoritmo de Dijkstra

- Toma como entrada un nodo X del grafo, y devuelve un arreglo de enteros, donde la posición i representa el mínimo costo para ir desde X hasta el vertice i.
- El vértice donde comienza lo llamaremos S, y el arreglo de distancias lo llamaremos *dist*.
- Utilizaremos una cola de prioridad (priority_queue) tal que sus elementos sean pares (distancia, vertice), y el orden de prioridad es de menor a mayor distancia. La llamaremos *pq*.
- Inicializar todas las posiciones de *dist* con infinito, asignar $dist[S] = 0$, e insertar el vértice S en pq con distancia 0.

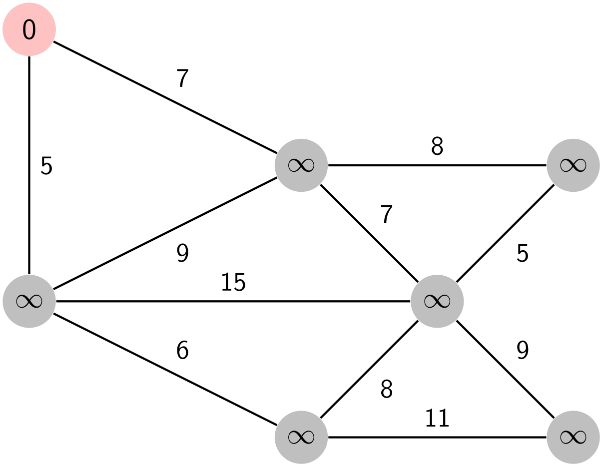
Algoritmo de Dijkstra

- Mientras que *pq* no esté vacía:
  - extraer el primer par (w, u) de la cola.
  - para cada vecino v de u, si $dist[u] + weight[u][v] < dist[v]$, cambiar $dist[v]$ por su nueva distancia y meterlo en la cola de prioridad.
- Al finalizar, el arreglo *dist* va a contener las distancias mínimas desde S hacia todos los vértices del grafo (si no existe camino desde S hacia X, $dist[X] = \text{infinito}$).

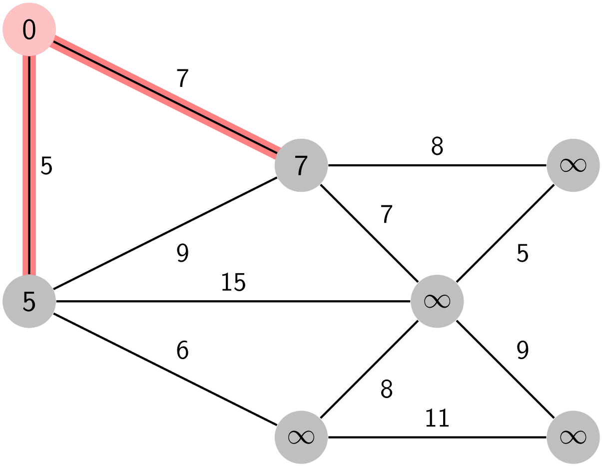
Algoritmo de Dijkstra

- La complejidad total es $O(n \log n)$
- El algoritmo no funciona si los pesos de los lados pueden ser negativos. Para ese tipo de grafos, el algoritmo de Bellman-Ford calcula distancias mínimas en $O(n * m)$, donde n es la cantidad de vértices y m es la cantidad de lados.

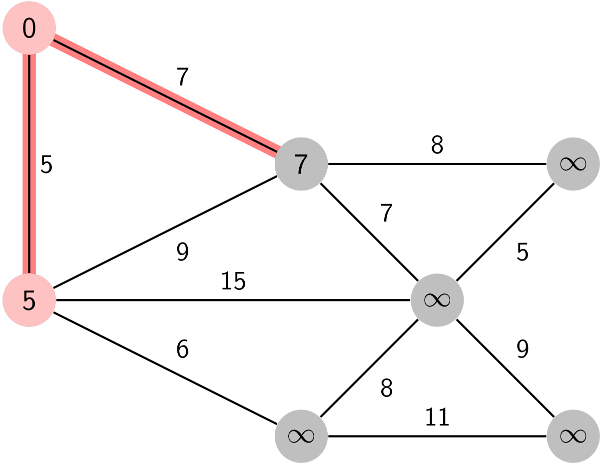
Ejemplo



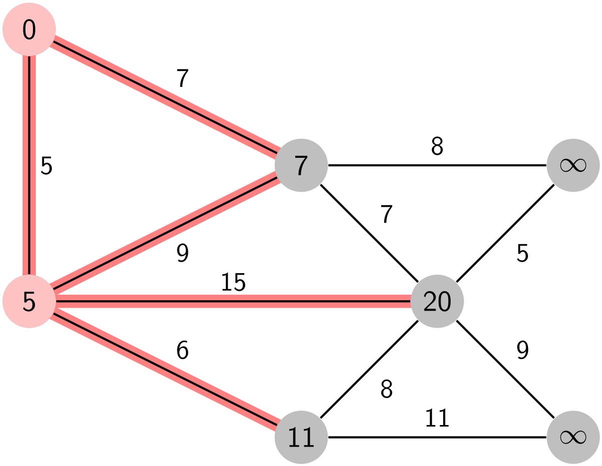
Ejemplo



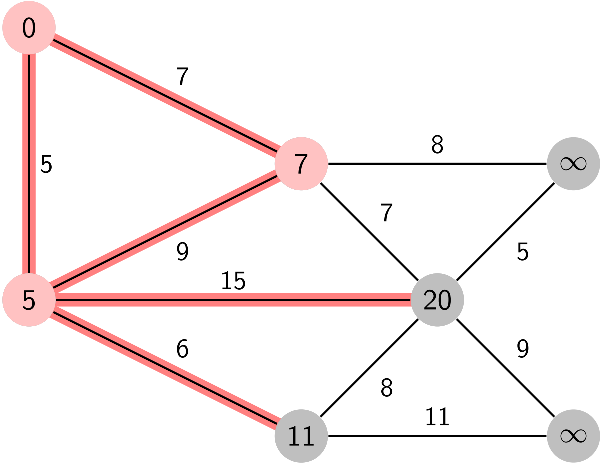
Ejemplo



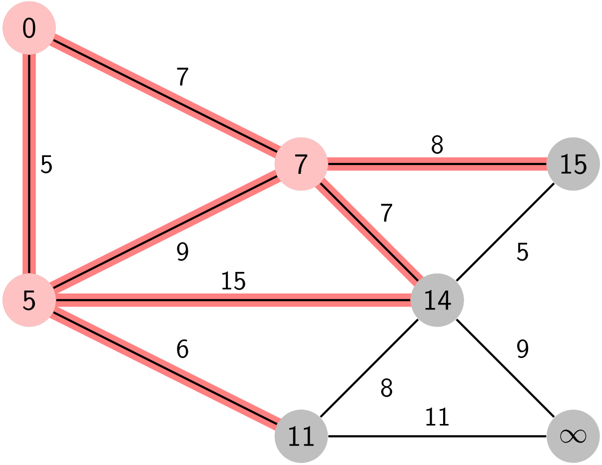
Ejemplo



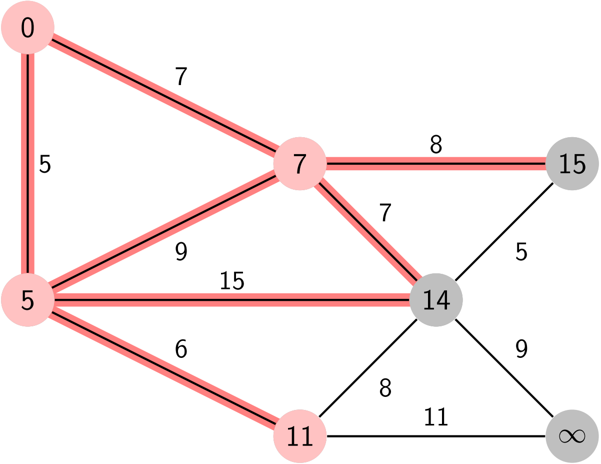
Ejemplo



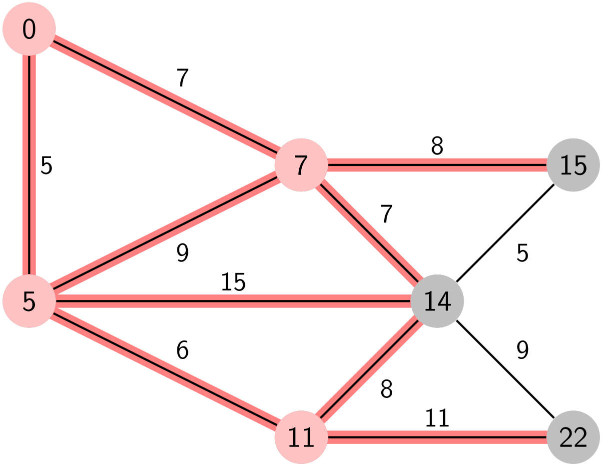
Ejemplo



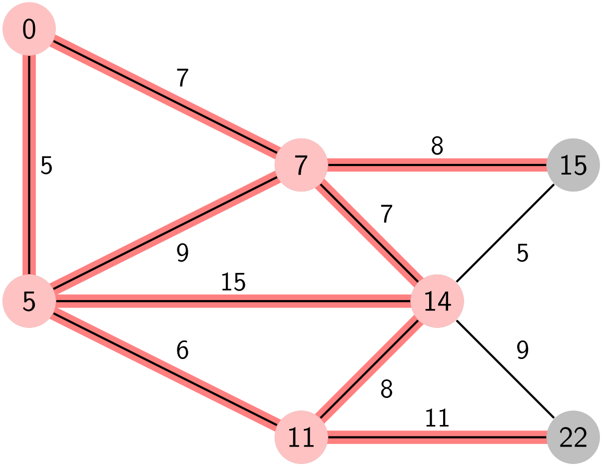
Ejemplo



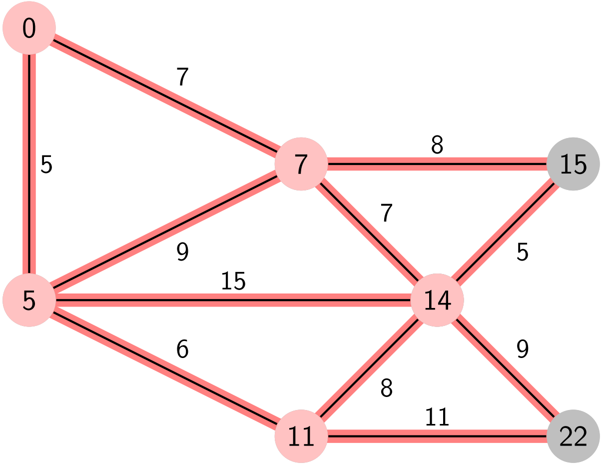
Ejemplo



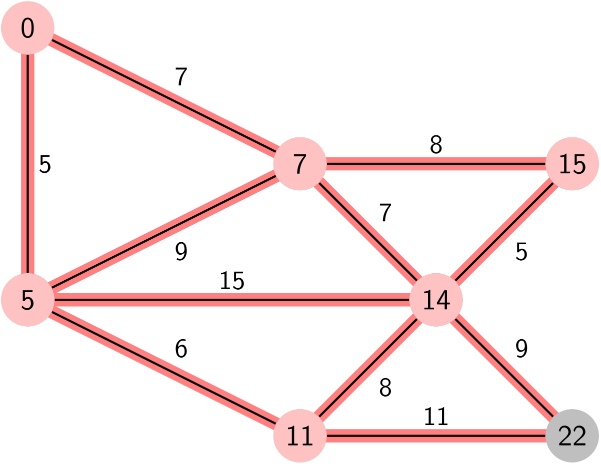
Ejemplo



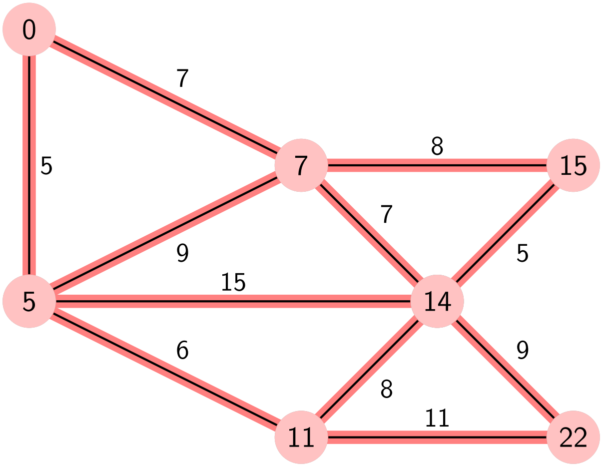
Ejemplo



Ejemplo



Ejemplo



Código de Dijkstra

```
void dijkstra(int src){
  pq.push({0, src});
  dist[src] = 0;
  while(!pq.empty()){
    int wh = pq.top().first;
    int u = pq.top().second;
    pq.pop();
    if(wh <= dist[u]){ //wh es la minima distancia hasta u?
      for(auto edges : graph[u]){
        int v = edges.second;
        int w = edges.first;
        if(dist[v] > dist[u] + w){
          dist[v] = dist[u] + w;
          pq.push({dist[v], v});
        }
      }
    }
  }
}
```

Algunos problemas de práctica

- http://codeforces.com/contest/510/problem/C
- http://www.spoj.com/problems/TOPOSORT
- http://codeforces.com/problemset/problem/20/C
- http://codeforces.com/problemset/problem/545/E
- http://codeforces.com/problemset/problem/938/D
- **Desafío de la clase (hay que manejar bien grafos dirigidos)**
  - http://codeforces.com/contest/919/problem/D