Laboratorio de Grafos

<nombre-de-su-ayudante><gen-de-su-ayudante>@udec.cl
05/06/2023

Motivación 💯

Piolin_Motivacion_128p.png

Para qué estudiamos/investigamos acerca de grafos?

- Por el simple hecho de estudiar/investigar.
- Como ejercicio intelectual/ocio (Euler y sus puentes).
- Para ampliar nuestras herramientas para solventar problemas.

Grafos del punto de vista ingenieril

- Herramienta para modelar problemas/situaciones.
- Nos permite aplicar todos los algoritmos y operaciones conocidas de grafos.
- No necesariamente podremos resolver un problema que podamos modelar como grafo (pero podemos intentarlo!)

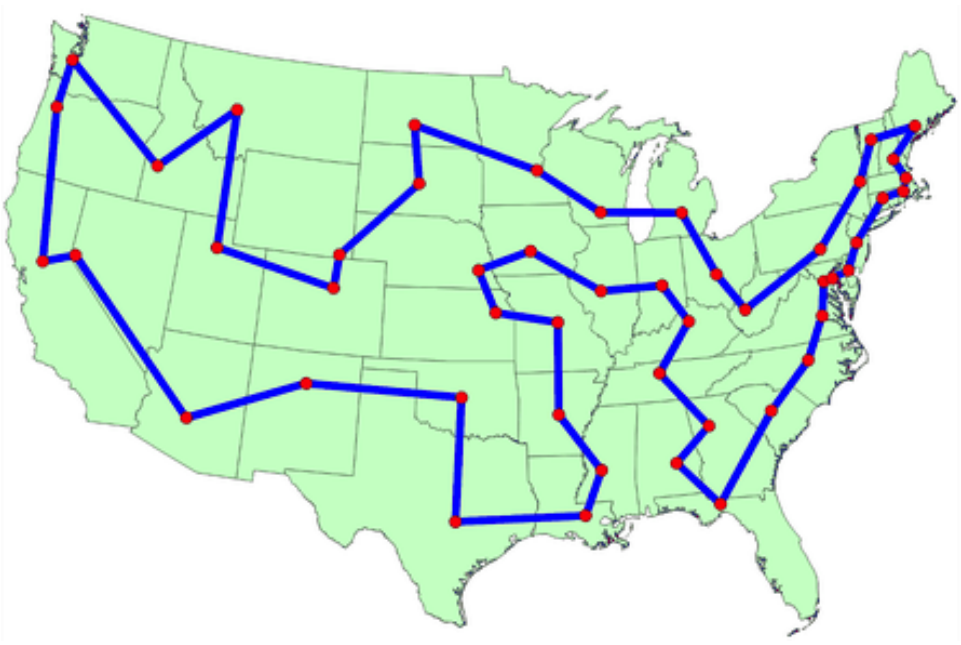


Figura 1: Travelling Salesman Problem, conocido problema NP-hard. Lo podemos modelar como un grafo, pero dificilmente lo vamos a resolver de esta forma.

Pregunta

Por qué razón podríamos descartar inmediatamente todos los algoritmos con complejidades temporales menores a exponencial para intentar resolver el TSP? 🤔

Grafo 🤓☝️

Un grafo G relaciona elementos de un conjunto (generalmente llamado V) entre ellos a través de 'conexiones' de pares de *vértices* o nodos.

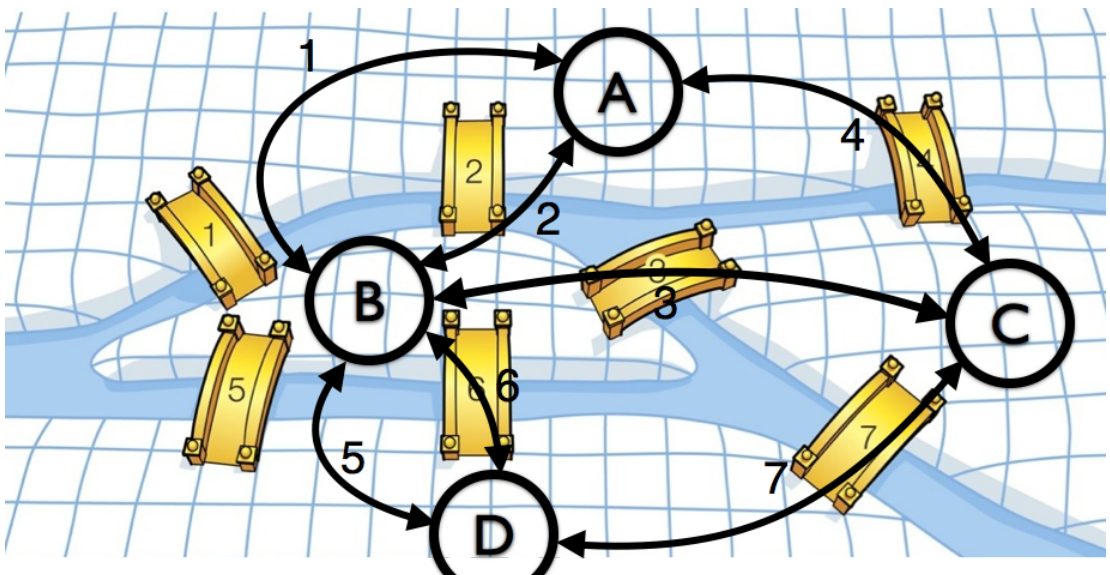


Figura 2: Puentes de Königsberg, el problema que dio pie a la teoría de grafos

$$G = (V, E)\ ,\ E \subset \{(u, v) : (u, v) \in V^2 \land u \neq v\}$$

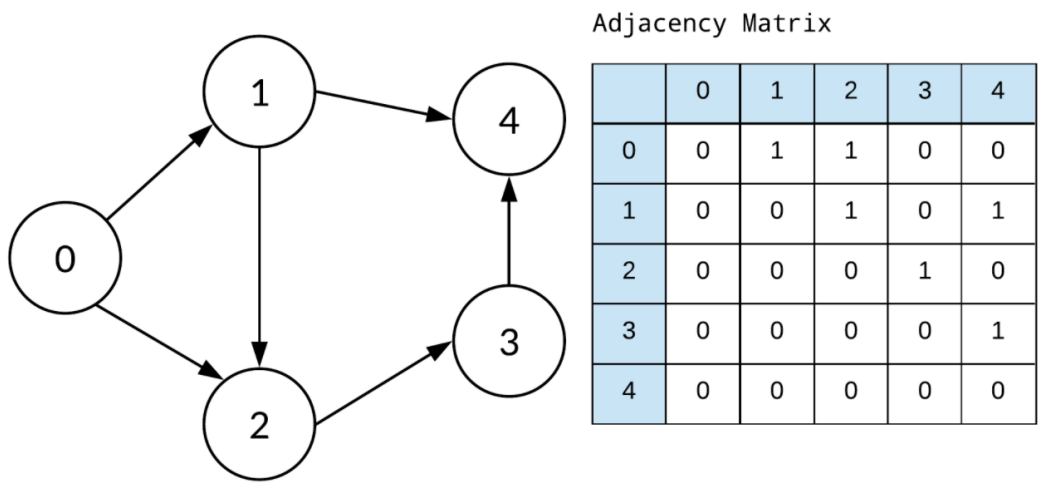
Presentaremos al mismo tiempo, 2 'versiones' distintas de grafo:

- Grafo no dirigido: todas las relaciones son simétricas $(u, v) \in E \iff (v, u) \in E$
- Grafo dirigido (digrafo): las relaciones no son necesariamente simétricas.
  Más allá de su representación como *estructura algebraica*, lo que más nos interesa es poder representarlo y utilizarlo al programar.

Representación de Grafos

Matriz de adyacencia

Complejidad espacial de $O(V^2)$



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

Figura 3: Representación de un grafo dirigido como matriz de adyacencia

Lista de adyacencia

Complejidad espacial de $O(V + E)$

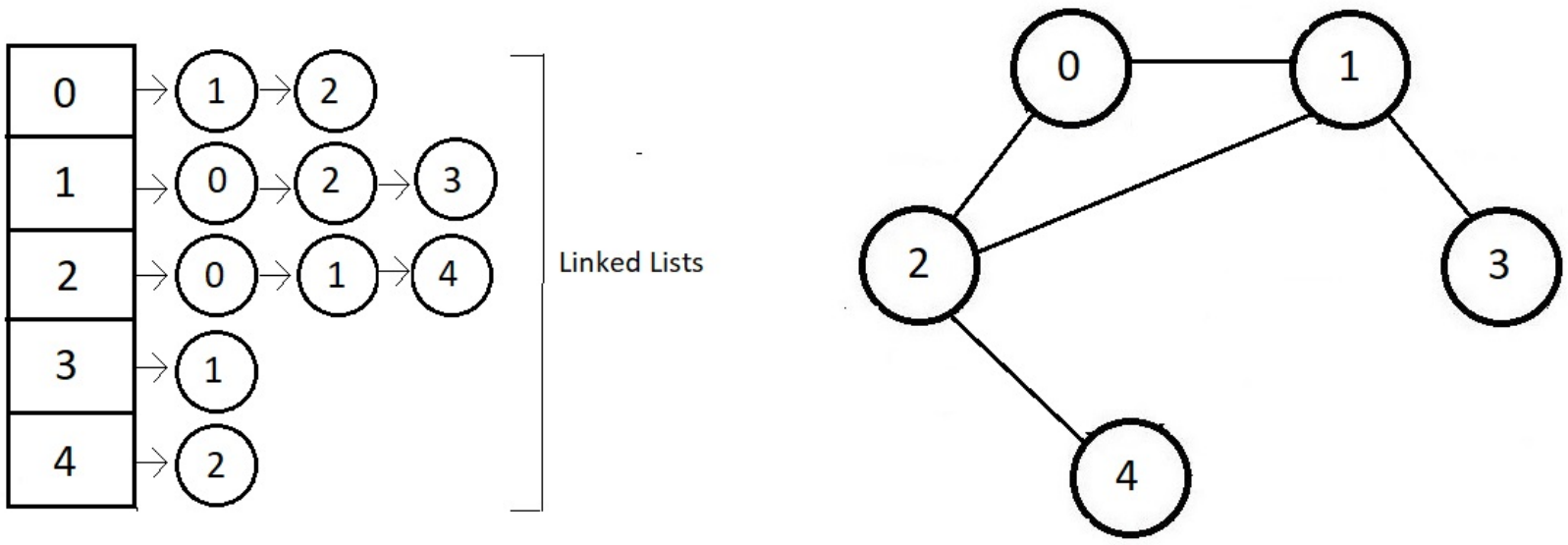


Figura 4: Representación de un grafo no dirigido como lista de adyacencia.

Operaciones

Solo con las representaciones ya podemos hacer algunas consultas!

- Conocer la *vecindad* de un vértice v.
- Saber si dos vértices están conectados.
- Listar todas las aristas (u, v) de un grafo.
  Cuál es la complejidad de estas operaciones en cada representación?

Implementación matriz de adyacencia

```
int matriz_adyacencia[NODOS][NODOS];

// Caso grafo no dirigido
void agregar_arista(int nodo1, int nodo2){
        matriz_adyacencia[nodo1][nodo2] = 1;
        matriz_adyacencia[nodo2][nodo1] = 1;
}

bool nodos_conectados(int nodo1, int nodo2){
        return matriz_adyacencia[nodo1][nodo2];
}

void listar_vecinos(int nodo){
        for(int i = 0; i < NODOS; i++)
                if(matriz_adyacencia[nodo][i] == 1)
                        cout << i << endl;
}
```

Implementación lista de adyacencia

```
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> adj_list(NODOS,vector<int>());

// Caso grafo dirigido
void agregar_arista(int nodo1, int nodo2){
        adj_list[nodo1].push_back(nodo2);
}

bool nodos_conectados(int nodo1, int nodo2){
        for(int nodo_vecino : adj_list[nodo1])
                if(nodo_vecino == nodo2) return true;
```

```
        return false;
}

void listar_vecinos(int nodo){
        for(int nodo_vecino : adj_list[nodo])
                cout<<nodo_vecino<<endl;
}

int main(){
        vector<pair<int, int>> lista_aristas;
        int a, b;
        for(int i = 0; i < cantidad_aristas; i++){
                cin>>a>>b;
                lista_aristas.push_back({a,b});
        }
        // imprimir primer arista
        cout<<lista_aristas[0].first<<" "<<lista_aristas[0].second<<endl;

        // Se ordena en base al elemento first
        sort(lista_aristas.begin(),lista_aristas.end());
        return 0;
}
```

Pregunta

Entonces, si la lista de adyacencia utiliza menos espacio, para que sirve la representación como matriz? 🤔

Recorrido de Grafos 🏃‍♀️

La mejor forma de entender estos recorridos, es visualizarlos!

- Visualizador: https://visualgo.net/en/dfsbfs

Ejercicio!

Una vez ya entendido los algoritmos de recorrido, remplace los siguientes pseudocódigos de *BFS* y *DFS* por código de C++.

BFS

```
Algorithm BFS(G, start):
  empty queue Q
  empty list DIST to keep track of the distance of the nodes

  Set start distance to 0
  Enqueue start into Q

  While Q is not empty:
    node = Dequeue from Q
    For each neighbour of node:
      If neighbour is INF in DIST:
        Enqueue neighbour into Q
        DIST[neighbour] = DIST[node] + 1;
```

DFS

Iterativo

```
Algorithm IterativeDFS(G, v):
  Initialize an empty stack S
```

```
  Push v onto S

  While S is not empty:
    Pop the top node t from S
    If t is not visited:
      Mark t as visited
      For each neighbour n of t in G:
        Push n onto S
```

Recursivo

```
Algorithm RecursiveDFS(G, v):
  Mark v as visited
  For each neighbour n of v in G:
    If n is not visited:
      Call RecursiveDFS(G, n)
```

Componentes Conexas y Fuertemente Conexas

Componente Conexa

Pregunta

Cómo podemos encontrar las componentes conexas de un grafo utilizando alguno de los algoritmos vistos anteriormente? 🤔

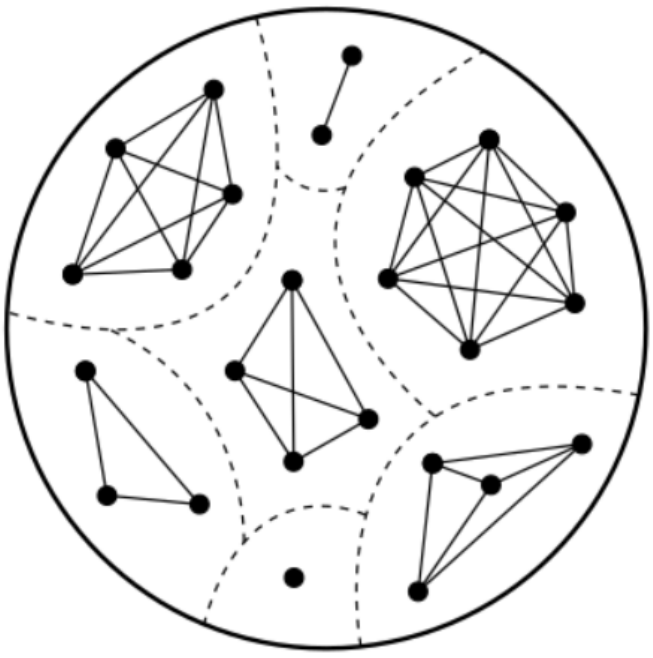


Figura 5: Grafo disconexo con 7 componentes conexas

Componentes Fuertemente Conexas

Se da solo en digrafos.
Corresponde a un subgrafo maximal donde para cada par de vértices en él, existe un *camino*.

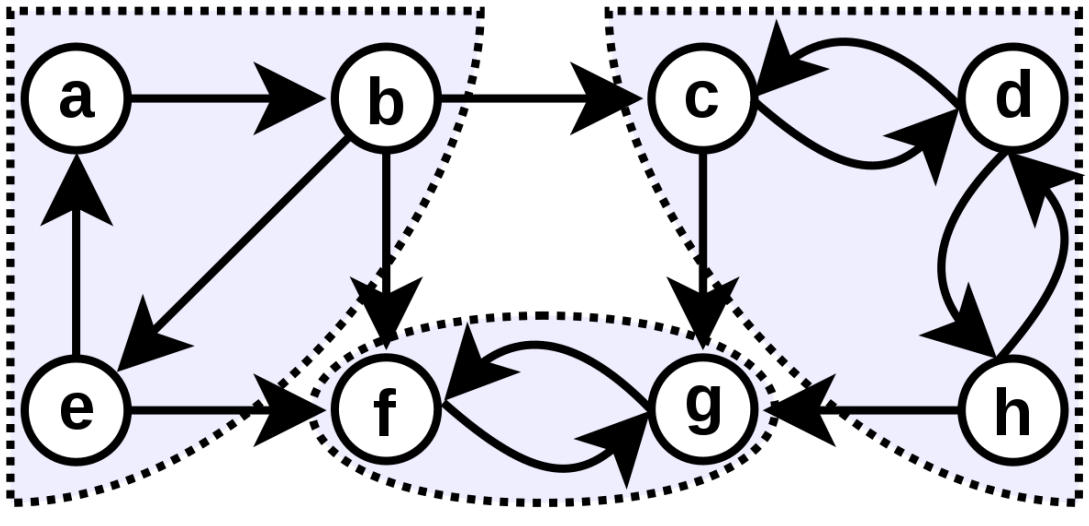


Figura 6: Digrafo con 3 componentes fuertemente conexas

Encontrar SCC: Algoritmo de Kosaraju

```
Algorithm Kosaraju(G):
  Initialize an empty stack S
  Mark all vertices as not visited

  For each vertex v in G:
    If v is not visited:
      Run DFS on v to fill the stack S

  Create a transposed graph G'

  Mark all vertices as not visited for the second DFS run

  While S is not empty:
    Get the top vertex v from S
    Pop v from S

    If v is not visited in the transposed graph:
      Run DFS on v in the transposed graph
      Each DFS run will give a strongly connected component

  End
```

Otra opción de algoritmo, y que también funciona con grafos con peso, es el algoritmo de Tarjan.

Grafos con peso 🏋️

Corresponden a grafos donde cada arista (u, v), tiene un peso asociado.

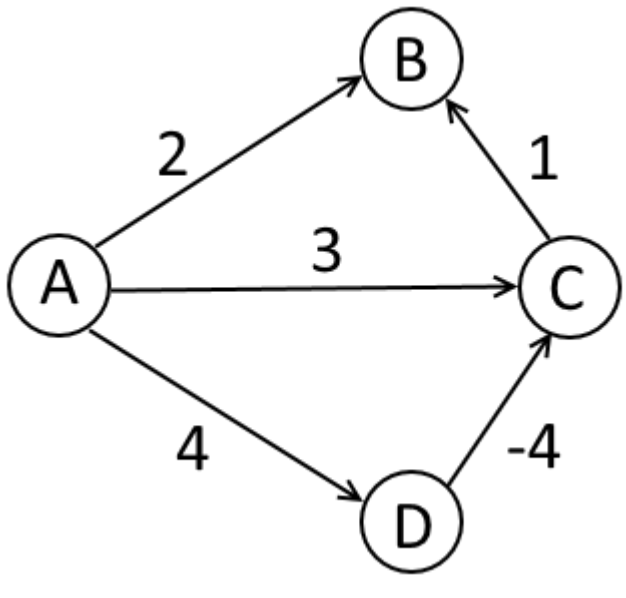


Figura 7: Grafo con peso

Encontrar camino mínimo: Algoritmo de Dijkstra

Considerar que solo funciona cuando todos los pesos del grafo son positivos (caso contrario se pueden aplicar los algoritmos de Bellman-Ford o Floyd-Warshall).

```
// https://github.com/stevenhalim/cpbook-code/blob/master/ch4/sssp/dijkstra.cpp
  priority_queue<ii, vector<ii>, greater<ii>> pq; pq.push({0, s});

  // sort the pairs by non-decreasing distance from s
  while (!pq.empty()){                      // main loop
    auto [d, u] = pq.top(); pq.pop();       // shortest unvisited u
    if (d > dist[u]) continue;              // a very important check
    for (auto &[v, w] : AL[u]) {            // all edges from u
      if (dist[u]+w >= dist[v]) continue;   // not improving, skip
      dist[v] = dist[u]+w;                  // relax operation
      pq.push({dist[v], v});                // enqueue better pair
    }
  }
```

```
  for (int u = 0; u < V; ++u)
    printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
```

Complejidad temporal de $O(E + V \log V)$.

Minimum Spanning Tree

Corresponde al subgrafo de un grafo que corresponda a un árbol, y tenga el menor valor posible.

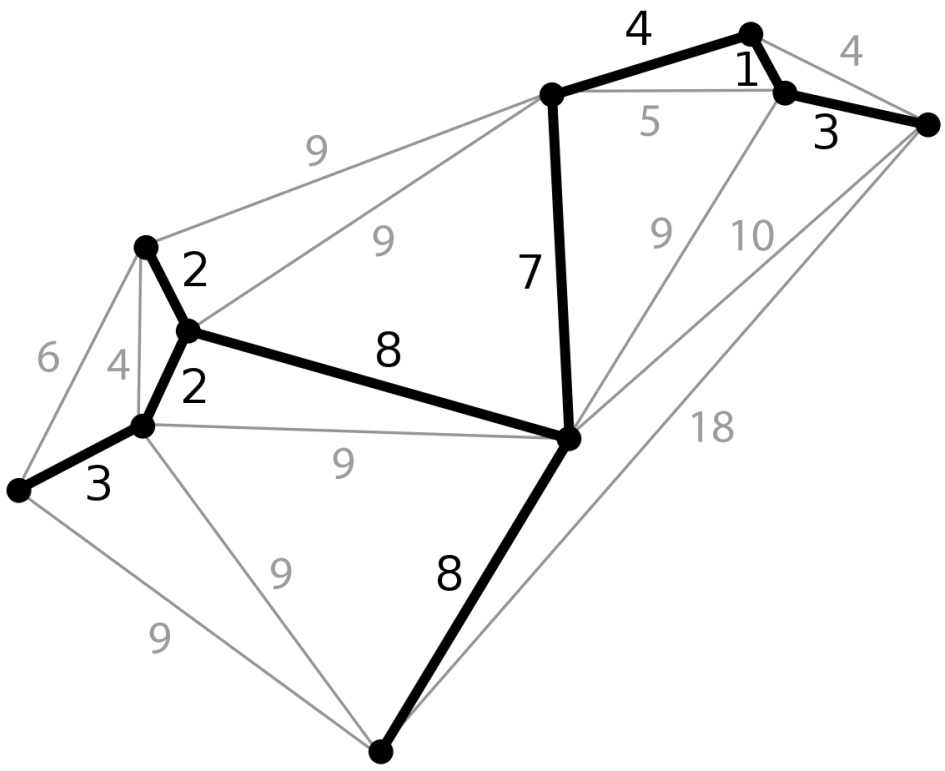


Figura 8: Minimum Spanning Tree de un grafo

Algoritmo de Kruskal

Idea principal

1. Ordena las aristas ascendentemente.
2. Se selecciona la arista con menor peso. Se incluye en el árbol si no forma un ciclo con las demás aristas que se han incluido anteriormente.
3. Repetir el proceso hasta que todos los nodos estén conectados.
   Para agregar nuevas aristas y verificar que no formen ciclos utiliza una (pequeña) estructura de datos auxiliar:

Union-Find Disjoint Sets

(El yunionfain para los amigos)

- $Union(x, y)$: Une dos conjuntos disjuntos que contengan los elementos x e y, en tiempo constante (amortizado).
- $Find(x)$: Encuentra el *representante* (no confundir con el padre) de un elemento x en tiempo constante (amortizado).
  El código es un poco largo como para mostrarlo aquí, pero está disponible en el siguiente repositorio.
  Complejidad temporal de $O(E \log E)$.
  Funciona mejor que Prim para grafos esparsos.

Algoritmo de Prim

Idea principal

1. El algoritmo de Prim comienza con un solo nodo en un grafo conexo con peso.
2. Expande progresivamente el árbol inicial añadiendo la arista de peso mínimo que conecta cualquier nodo dentro del árbol con un nodo fuera del árbol.
3. Este proceso de adición de aristas continúa hasta que todos los nodos están incluidos en el árbol.
4. El resultado de este proceso es un MST.

Implementación

```
// https://github.com/stevenhalim/cpbook-code/blob/master/ch4/mst/prim.cpp

// inside int main() --- assume the graph is stored in AL, pq is empty
  int V, E; scanf("%d %d", &V, &E);
  AL.assign(V, vii());
  for (int i = 0; i < E; ++i) {
    int u, v, w; scanf("%d %d %d", &u, &v, &w);  // read as (u, v, w)
    AL[u].emplace_back(v, w);
    AL[v].emplace_back(u, w);
  }
  taken.assign(V, 0);                   // no vertex is taken
  process(0);                           // take+process vertex 0
  int mst_cost = 0, num_taken = 0;      // no edge has been taken
  while (!pq.empty()) {                 // up to O(E)
    auto [w, u] = pq.top(); pq.pop();   // C++17 style
    w = -w; u = -u;                     // negate to reverse order
    if (taken[u]) continue;             // already taken, skipped
    mst_cost += w;                      // add w of this edge
    process(u);                         // take+process vertex u
    ++num_taken;                        // 1 more edge is taken
    if (num_taken == V-1) break;        // optimization
  }
  printf("MST cost = %d (Prim's)\n", mst_cost);
```

Notar que posee cierta similitud con el algoritmo de Dijkstra.
La forma más usual en la que se implementa este algoritmo (Binary Heap + Adj List) tiene una complejidad temporal de $O(E \log V)$.
Funciona mejor que Kruskal para grafos densos.

Densidad de un Grafo

Ya en dos ocasiones la elección sobre que herramienta usar depende de que tan denso o esparso sea el grafo con el que estemos trabajando (*Lista* vs *Matriz*, y *Kruskal* vs *Prim*), por lo que a continuación procedemos a definir esta propiedad:

$$m = \frac{n(n-1)}{2} : \text{cantidad máxima de aristas para un grafo no dirigido}$$

$$\Delta = \frac{2m}{n(n-1)} : \text{densidad de un grafo no dirigido}$$

$$\Delta = \frac{m}{n(n-1)} : \text{densidad de un grafo dirigido}$$

Otras opciones para seleccionar alguna representación o algoritmo de grafos, son calcular las complejidades de lo que se vaya a utilizar en base a las cardinalidades de los conjuntos que tengamos (en el caso de estar disponibles). Ej: Si $|V| = 10,\ |E| = 5 \implies O(|V| + |E|) = O(15)$.
También, en base a la "naturaleza" del grafo, o del enunciado del problema que estemos trabajando, podemos tener una idea de la densidad. Ej: "Un grafo donde los vértices son países y las aristas representan países vecinos. Entonces sabemos que existen muchos vértices y pocas aristas (grafo esparso)".

Conclusión 🧚✨

Le dimos una pincelada a la implementación de varios de los conceptos fundamentales de los grafos y digrafos en el contexto de estructuras de datos.
Sin embargo, existen una infinidad de otros conceptos, algoritmos, representaciones de esta estructura, por lo que los invitamos a investigar más por su cuenta en el caso de estar interesados :).