

Grafos

Definiciones básicas

- Los **grafos** son un conjunto de vertices y ejes $G = (V, E)$
- si no aclara no tiene dirección
- No hay selfloops ni varios ejes para un mismo par de nodos (multigrafos)

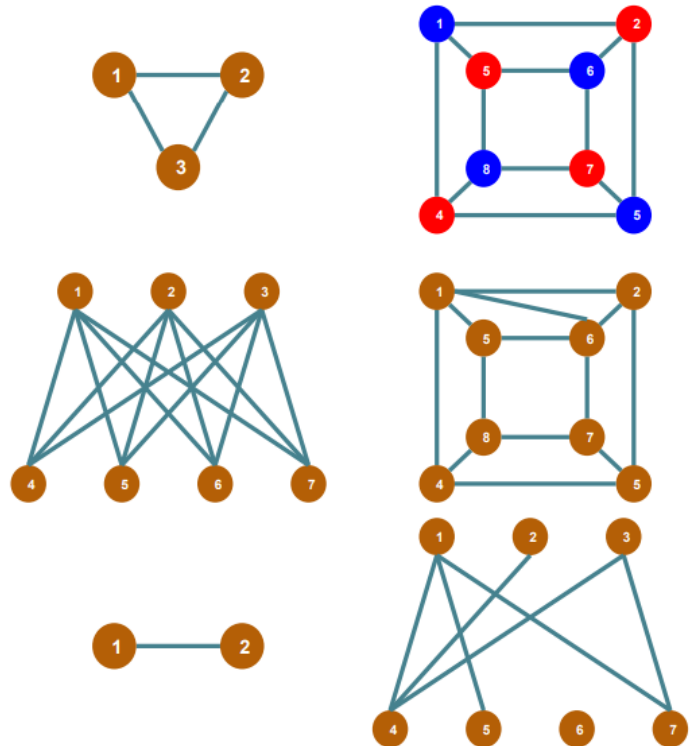
Tipos de grafos

- **Digrafo:** grafo con ejes con dirección (a lo sumo puede tener 2 ejes por par de nodos, uno para la ida y otro para la vuelta)
 - $d_{in}(v)$: grado de entrada de un nodo
 - $d_{out}(v)$: grado de salida de un nodo
 - las aristas E ahora son pares ordenados donde, sea $e=(u, v)$, u se lo llama arco, u cola (desde donde viene) y v cabeza (hacia donde va)
- Dos grafos son isomorfismo si son iguales salvo por el nombre de los nodos (los rotas y quedan igual en distribución)
- **Grafos completos:** (para grafos sin dirección), todos los nodos estan conectados con todos
- **Grafo complemento:** Es el grafo con el mismo conjunto de vertices pero solo tiene las aristas que NO estan en G .
- **Grafo conexo:** (para grafos sin dirección) si existe un camino para todo par de vertices
- **Grafo fuertemente conexo:** (directed graphs) si existe un camino orientado entre todo par de vertices.
- **Grafos bipartitos:** es un grafo cuyos vértices se pueden separar en dos conjuntos disjuntos, de manera que las aristas no pueden relacionar vértices de un mismo conjunto.1

Grafos bipartitos

Definición 12:

- Un grafo $G = (V, E)$ es **bipartito** si existen dos subconjuntos V_1 y V_2 de V tal que
 - $V = V_1 \cup V_2$
 - $V_1 \cap V_2 = \emptyset$.
 - Para todo $e=(u,v) \in E$, $u \in V_1$ y $v \in V_2$
- Un grafo $G = (V, E)$ es **bipartito completo** con particiones V_1 y V_2 , si además todo vértice en V_1 es adyacente a todo vértice en V_2 .



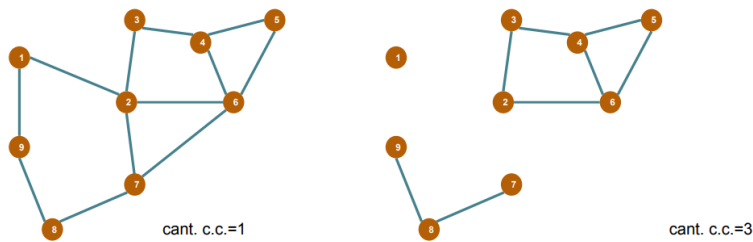
- **Grafos bipartitos completos:** es un grafo bipartito en que todos los vértices de uno de los subconjuntos están relacionados con los del otro subconjunto

- **subgrafos:**

Definición 9: Subgrafos:

- Dado un grafo $G = (V_G, E_G)$, un **subgrafo** de es un grafo $H = (V_H, E_H)$ tal que $V_H \subseteq V_G, E_H \subseteq E_G \cap (V_H \times V_H) \Rightarrow H \subseteq G$
- Si $H \subseteq G$ y $H \neq G \Rightarrow H$ es un **subgrafo propio** de G
- Si $H \subseteq G$ y $V_H = V_G \Rightarrow H$ es un **subgrafo generador** de G
- Si $H \subseteq G$ y para todo $u, v \in V_H$ con $e = (u, v) \in E_G$ entonces también vale $e = (u, v) \in E_H$ (tiene todas las aristas que conectan los vértices de H y están en G) $\Rightarrow H$ es un **subgrafo inducido** de G . H se lo nota $V_{[G]}$

- **Componente conexa:** es un subgrafo conexo maximal (no está incluido estrictamente en otro grafo).



- **Arboles:**
 - Grafo conectado y aciclico
 - si saco una arista, se desconecta
 - si agrego una arista se forma un ciclo

caminos y recorridos

- **Vecindario:** $N(v)$ es el conjunto de nodos de G adyacentes a v
- **grado:** $d(v) = |N(v)|$ es la cantidad de vecinos
- **recorrido:** una secuencia de nodos conectados por aristas (pueden repetir nodos)
- **longitud de un recorrido:** la cantidad de *aristas* que tiene
- **Distancia** entre 2 nodos: el camino mas corto $d(u, v)$. Si no existe es ∞ . La distancia de un vertice con si mismo es 0.
- **camino:** un recorrido sin nodos repetidos (tambien se llama camino simple cuando se refiere al recorrido como camino)
- **circuito:** un recorrido que empieza y termina en el mismo nodo
- **ciclo:** un circuito que no repite nodos (se puede decir ciclo al circuito y circuito simple al circuito)
- No es valido un ciclo de longitud 2
- **punte:** es una arista que al quitarla aumenta la cantidad de componentes conexas del grafo

Isomorfismo de grafos:

Definición 13: Isomorfos

Dados $G=(V,E)$ y $G'=(V',E')$ son **isomorfos** si existe una función biyectiva $f: V \rightarrow V'$ tq para todo $u,v \in V$:

$$(u,v) \in E \Leftrightarrow (f(u),f(v)) \in E'$$

f : función de isomorfismo, $G = G'$ (abuso de notación)

Proposición 3: Isomorfos

Si dos grafos $G=(V,E)$ y $G'=(V',E')$ son **isomorfos**, entonces,

1. tienen el mismo número de vértices
2. tienen el mismo número de aristas
3. para todo k , $1 \leq k \leq n-1$, tienen el mismo número de vértices de grado k (la misma distribución de grado)
4. tienen el mismo número de c.c.
5. para todo k , $1 \leq k \leq n-1$, tienen el mismo número de caminos simples de longitud k .

Teorema 1: suma de grados

$$\sum_{v \in V} d(v) = 2m$$

siendo m la cantidad de aristas del grafo

colorario : la cantidad de nodos con grado impar es par. Esto debido a que la suma de todas las conexiones da un numero impar. Si sumamos 2 pares da par, si sumamos 2 impares da par, por lo que para preservar la paridad tiene que haber una cantidad par de nodos con grado impar.

DFS y BFS

DFS sin stack

```
vector<vector<int>>> aristas = ... ;  
vector<bool> visitado(n, false);
```

```
void dfs(int v) {  
    visitado[v] = true;  
  
    for (int u : aristas[v]) {  
        if (!visitado[u]) {  
            dfs(u)  
        }  
    }  
}
```

DFS con stack

```
vector<vector<int>>> aristas = ... ;  
vector<bool> visitado(n, false);
```

```
void dfs(int v) {  
    stack<int> s;  
    s.push(v);  
    visitados[v] = true  
  
    while (!s.empty()) {  
        int v = s.top(); s.pop();  
  
        for (auto u: aristas[v]) {  
            if (!visitado[u]) {  
                visitado[u] = true;  
                s.push(u);  
            }  
        }  
    }  
}
```

```

    }
}

}

}

```

BFS con cola

```

vector<vector<int>> aristas = ...;
vector<bool> visitados(n, false);
vector<int> distancia(n);

void bfs(int s) {
    visitados[s] = true;
    distancia[s] = 0;

    queue<int> q;
    q.push(s);

    while (!q.empty()) {
        int v = q.front(); q.pop();

        for (auto u : aristas[v]) {
            if (!visitado[u]) {
                visitado[u] = true;
                distancia[u] = distancia[v] + 1;
                q.push(u);
            }
        }
    }
}

```

Problemas que se resuelven con DFS y BFS

1. **¿Es el grafo conexo?:** Recorro con DFS o BFS desde algún nodo (el 0 por ejemplo) y si el vector de visitados no tiene ningún falso, entonces sí es conexo. De lo contrario hay nodos que desde el que comenzamos no podemos llegar.
2. **¿Cuántas componentes conexas tengo?:** Cada vez que recorro el grafo desde un vertice, me marca como visitado los que pertenecen a su componente conexas. Luego me fijo cuales no fueron marcadas, recorro desde la primera no marcada para encontrar todos los elementos de la otra componente conexas. Repito para obtener todas las componentes conexas.

```

int cant_comp_conexas = 0;
for (int i = 0; i < n; i++) { // n la cantidad de vertices del grafo
    if (!visitado[i]) {
        cant_comp_conexas++;
        recorro_desde_el_vertice(i); // con dfs o bfs
    }
}

```

3. **¿Hay ciclos? Y si hay, guardame alguno:** Recorro el grafo y en cada paso guardo el padre del nodo actual. Recorro los vecinos del nodo actual v. Para cada vecino u, si no está recorrida le pongo el padre y la recorro, si está recorrida y no es el padre de v, entonces se hay un ciclo desde v hasta ese u

```

vector<int> padres(n, -1);
vector<int> ciclo;
vector<vector<int>> aristas = ...;
int comienzo_ciclo = -1, fin_ciclo = -1;

```

```

queue<int> q;

for (int i = 0; i < n && comienzo_ciclo == fin_ciclo == -1; i++) {
    if (padre[i] == -1) {
        dfs_ciclos(i);
    }
}

if (comienzo_ciclo >= 0) {
    int v = comienzo_ciclo;
    ciclo.push_back(v);

    while (v != fin_ciclo) {
        v = padre[v];
        ciclo.push_back(v);
    }
}

void dfs_ciclos(int s) {
    padre[s] = 0;
    q.push(s);

    while (!q.empty() && comienzo_ciclo == fin_ciclo == -1) {
        int v = q.top(); q.pop();

        for (auto u : aristas[v]) {
            if (padre[u] == -1) {
                padre[u] = v;
                q.push(u);
            } else if (u != padre[v]) {
                //si la arista no es la que apunta al padre de v, ciclo.
                comienzo_ciclo = v;
                fin_ciclo = u;
                break;
            }
        }
    }
}

```

4. **¿Es un grafo bipartito?:** Recorro el grafo pintando de 2 colores, cuando me encuentro en el recorrido uno del mismo color que su padre, entonces no es bipartito

```

void dfs_bipartito(int v) {
    for (int u : aristas[v]) {
        if (color[u] == -1) {
            // si no esta pintado pinto
            color[u] = 1 - color[v];
            dfs(u);
        } else if (color[u] == color[v]) {
            // si ya esta pintado chequeo
            es_bipartito = false;
        }
    }
}
// llamada

```

```

es_bipartito = true;
color[0] = 0;
dfs_bipartito(0);

```

5. **Cantidad de caminos desde w a v:** Luego de hacer BFS en w, tengo computadas todas las distancias desde w hacia cualquier nodo que tenga un camino con w. Luego para buscar la cantidad de caminos, cuento la cantidad de caminos que hay desde los vecinos con distancia menor a v. Cuando la distancia es 0, estoy en el nodo w y hay un solo camino.

```

BFS(w)
cantidad_de_caminos_hasta(v);

int cantidad_de_caminos_hasta(v) {
    if (distancia[v] == 0) return 1;
    if (memo[v] != -1) return memo[v];
    int res = 0;

    for (int vecino : aristas[v]) {
        if (distancia[vecino] + 1 == distancia[v]) {
            res += cantidad_de_caminos_hasta(vecino);
        }
    }

    memo[v] = res;
    return res;
}

```

6. **Cantidad de puentes de un grafo:** Un puente es una arista que al quitarla aumenta la cantidad de componentes conexas del grafo. En el recorrido DFS puedo tener 3 estados para cada vertice: Empecé a recorrer, terminé de recorrer, no lo recorrí. Según estos estados podemos clasificar a las aristas según el árbol generado por el DFS. Dado un momento del recorrido en un vertice V, vamos a ver todos sus vecinos. Si el vecino no lo recorrí, entonces la arista es un tree-edge (pertenece al árbol DFS). Si no, y si no es el padre, entonces es un back-edge (conecta con un ancestro) y no pertenece al árbol DFS. Una back-edge no es un puente. Una arista es un puente si es un tree-edge y no tiene una back-edge “que la cubra”. La cantidad de back-edges que cubre a una arista v de su padre es $\sum_{w \text{ hijo de } v} \text{cubren}(w) - \text{backEdgesQueTerminanEn}(v) + \text{backEdgesQueEmpiezanEn}(v)$. $\text{backEdgesQueTerminanEn}(v)$: aristas que no están en el DFS que empiezan en algún descendiente de v y que terminan conectado en v. $\text{backEdgesQueEmpiezanEn}(v)$: aristas que empiezan en v y van hacia algún ancestro de v.

- **Complejidad:** $O(n + m)$

```

vector<int> memo(n, -1);
int NO_LO_VI = 0, EMPECE_A_VER = 1, TERMINE_DE_VER = 2;
vector<int> estado(n, NO_LO_VI);
vector<vector<int>> tree_edges(n);
vector<int> back_edges_con_extremo_inferior_en(n), back_edges_con_extremo_superior_en(n);

void dfs_puentes(int v, int p = -1) {
    estado[v] = EMPECE_A_VER;
    for (int u : aristas[v]) {
        if (estado[u] == NO_LO_VI) {
            tree_edges[v].push_back(u);
            dfs_puentes(u, v);
        } else if (u != p) {
            if (estado[u] == EMPECE_A_VER)
                back_edges_con_extremo_inferior_en[v]++;
            else // estado[u] == TERMINE_DE_VER
                back_edges_con_extremo_superior_en[v]++;
        }
    }
    estado[v] = TERMINE_DE_VER;
}

```

```

}

int cubren(int v, int p = -1) {
    if (memo[v] != -1) return memo[v];
    int res = 0;

    for (int hijo : tree_edges[v]) {
        if (hijo != p) {
            res += cubren(hijo, v);
        }
    }

    res -= back_edges_con_extremo_superior_en[v];
    res += back_edges_con_extremo_inferior_en[v];
    memo[v] = res;
    return res;
}

int componentes = 0, cant_puentes = 0;

for (int i = 0; i < n; i++) {
    if (estado[i] != NO_LO_VI) {
        dfs_puentes(i);
        componentes++;
    }
}

for (int i = 0; i < n; i++) {
    if (cubren(i) == 0){
        cant_puentes++;
    }
}

// Por cada componente conexa hay una raiz que no lo cubre nadie,
// pero no cuenta para aristas puente
cant_puentes -= componentes

```

7. **Topological sort:** Dado un grafo acíclico dirigido G , es una ordenación lineal de todos los nodos de G que satisface que si G contiene la arista dirigida uv entonces el nodo u aparece antes del nodo v . En grafos con ciclos no hay orden topológico. (Se ordenan los nodos en orden de precedencia)

```

vector<lista<int>> aristas = ...;
vector<bool> visitado(bool, false)
vector<int> vertices_ordenados;
stack<int> res;

for (int i = 0; i < n; i++) {
    if (!visitado[i]) {
        dfs_topological_sort(i);
    }
}

for(int i = 0; i < n; i++) {
    vertices_ordenados.push_back(res.pop());
}

void dfs_topological_sort(int v) {
    visitado[v] = true;

    for (auto u : aristas[v]) {

```

```

    if (!visitado[u]) {
        dfs_topological_sort(u);
    }
}
res.push(v);
}

```

Arboles

Definiciones

- **Árbol:** T (grafo conexo sin circuitos simples)
- **Hoja:** u $d(u) = 1$ (un vertice con grado 1)
- **Raíz:** Algún vértice elegido
- **Bosque:** Conjunto de árboles (componentes conexas)
- **Árbol trivial:** T con $n=1$ y $m=0$

Equivalencias (cada una implica la otra)

1. G es un árbol (grafo conexo sin circuitos simples).
2. G es un grafo sin circuitos simples y e una arista $tq\ e \notin E$. $G+e = (V, E+\{e\})$ tiene exactamente un circuito simple, y ese circuito contiene a e . Es decir, si agrego una arista cualquiera se forma un ciclo.
3. \exists exactamente un camino simple entre todo par de nodos.
4. G es conexo, pero si se quita cualquier arista queda un grafo no conexo. Es decir, si saco cualquier arista se desconecta, o toda arista es puente

Propiedades

- **Lema 1:** La unión entre dos caminos simples distintos entre u y v contiene un circuito simple.
- **Lema 2:** Sea $G = (V, E)$ un grafo conexo y $e=(v,u) \in E$. $G - e = (V, E - \{e\})$ es conexo $\iff e \in C$: circuito simple de G . ($e=(v,u) \in E$ es puente $\iff e$ no pertenece a un circuito simple de G).
- **Lema 3:** Todo árbol no trivial tiene al menos dos hojas.
- **Lema 4:** Sea $G = (V, E)$ un árbol $\Rightarrow m = n - 1$ ($m = |E|$ $n = |V|$) (dem: inducción en n , sacas una hoja y la vuelves a poner)
- **Colorario 1:** Sea G un *bosque* con c c.c. $\Rightarrow m = n - c$
- **Colorario 2:** Sea G un *grafo* con c c.c. $\Rightarrow m \geq n - c$

Mas equivalencias (teorema 2?)

1. G es un árbol (grafo conexo sin circuitos simples).
2. G es un grafo sin circuitos simples y $m = n - 1$
3. G es un grafo conexo y $m = n - 1$

Árbol generador

Un árbol generador (AG) de un grafo G es un subgrafo que tiene el mismo conjunto de vértices y es un árbol

Teorema 4: 1. Todo G conexo tiene al menos un AG. 2. Si G conexo tiene un sólo AG entonces es un árbol. 3. $T=(V, E_T)$ es AG de $G=(V, E)$. Sea $e \in E - E_T$ (no está en el árbol) $tq\ T' = T+e-f = (V, E \cup \{e\} - \{f\})$ con f una arista del único circuito que se forma al agregar e (de $T+e$) $\Rightarrow T'$ es otro AG de G .

Árbol generador mínimo

Dado un grafo $G=(V, E, w)$ con $w : E \rightarrow R$ una función de costo para cada arista - Costo del AGM T : $w(T) = \sum_{e \in E(T)} w(e)$ (la suma de los costos de las aristas del AGM) - AGM es el AG para el cual $\sum_T w$ es mínima. - Para los grafos no pesados todo AG es AGM porque $w=1 \Rightarrow \sum_T w = m = n - 1$ - También puede haber varios AGM. Prim y Kruskal son 2 algoritmos para obtener AGMs

Prim

- *Invariante*: tenemos un árbol de i aristas que es subgrafo de algún AGM. *Inicialización*: empezamos con un solo vértice v arbitrario. *Iteración*: agregamos, de las aristas que podemos agregar y seguir teniendo un árbol, la más barata (Algoritmo goloso)
- **complejidad**: Hay implementaciones en $O(m * \log(n))$ y $O(n^2)$

```
vector<int> prim(int raiz, vector<lista<int>> ady) {
    vector<int> costo_vertices(n, inf);
    vector<int> padres_vertices(n, -1);
    vector<bool> elementos_enCola(n, true);
    // los vertices se numeran de 1..n
    padres_vertices[raiz] = 0;
    costo_vertices[raiz] = 0;

    // encollo todos los elementos en un min heap,
    // cada elemento es un par <costo, elemento>
    min_heap<pair<int, int>> Q; // el algoritmo de heapify es lineal, O(V)
    for (int i = 1; i <= n; i++) {
        Q.insert(pair<int, int>(inf, i));
    }
    // O(V*log(V) + E*log(V)) por cada vertice, desencolo una vez, +, por cada arista
    // (en el peor caso) hago un cambio en la cola de prioridad
    while (!Q.empty()) {
        int w = Q.getMin(); Q.extractMin();
        elementos_enCola[w] = false;

        for (auto v : ady[w]) {
            /* cuando agrego una arista al arbol generado en la iteración actual,
            éste puede tener aristas hacia algún nodo que ya estaba en el arbol
            pero con un costo menor.
            A--2--B
            |  /
            4 1
            | /
            C
            Si empezamos en A, agregamos las aristas A-B y A-C. Luego en el tope de la cola
            queda B y vemos que tiene una arista B-C que tiene un coste menor que lo
            computado hasta ahora (4 > 1), por lo que lo actualizamos y obtenemos el AGM con costo 3
            */
            if (elementos_enCola[v] && costo_vertice(v) > costo(arista(w,v))) {
                costo_vertice[v] = costo(arista(w,v));
                padres_vertices[v] = w;
                Q.decreaseKey(pair<int,int>(costo_vertice[v], v));
            }
        }
    }
    return padres_vertices;
}
```

- **complejidad**: $O(V + V * \log(V) + E * \log(V)) = O(E * \log(V)) = O(m * \log(n))$. Si el grafo tiene una cantidad mayor o igual de aristas que de vértices y usando binary heap y lista de adyacencias. Se puede mejorar con fibonacci heap a $O(m + n * \log(n))$.
- **Implementación** $O(n^2)$: consiste en hacer n veces buscar la arista (de las que todavía no se agregaron) con mínimo costo a agregar. (la cátedra dió una implementación)

Kruskal

- Consiste en ordenar todas las aristas de menor a mayor según la función de w (peso o costo). Luego, inicialmente, todos los elementos forman su propia componente conexa y voy uniendolos recorriendo en orden de menor a mayor costo las aristas del grafo. Las agrego al bosque si no generan un ciclo, es decir si sus extremos no pertenecen a la misma cc.
 - *Invariante*: tenemos un bosque generador de i aristas que es subgrafo de alg' un AGM.
 - *Invariante alternativo*: tenemos un bosque generador de i aristas que es mínimo entre los bosques de i aristas.
 - *Obs*: El primer invariante no implica al invariante alternativo. Pensar un ejemplo.
 - *Inicialización*: empezamos con todos los v'ertices y ninguna arista.
 - *Iteración*: agregamos, de las aristas que podemos agregar y seguir teniendo un bosque (no generan ciclos), la m'as barata
- **complejidad**: Hay implementaciones en $O(m * \log(n))$ y $O(n^2)$. La siguiente es $O(m * \log(n))$

```
struct DSU {
    DSU(int n){
        padre = vector<int>(n);
        for(int v = 0; v < n; v++) padre[v] = v;
        tamano = vector<int>(n,1);
    }
    // dado un v'ertice me devuelve a qu' componente conexa pertenece
    int find(int v){
        while(padre[v] != v) v = padre[v];
        return v;
    }
    // une las componentes conexas de u y v
    void unite(int u, int v){
        u = find(u); v = find(v);
        if(tamano[u] < tamano[v]) swap(u,v);
        //ahora u es al menos tan grande como v
        padre[v] = u;
        tamano[u] += tamano[v];
    }

    vector<int> padre;
    vector<int> tamano;

    //tamano[v] <= n
    //INV: si padre[v] != v entonces tamano[padre[v]] >= 2*tamano[v]
};

void kruskal(vector<tuple<int,int,int>>& E, int n){
    long long res = 0;
    sort(E.begin(),E.end());
    DSU dsu(n);

    int aristas = 0;
    for(auto [w,u,v] : E){
        //u y v estan en distinta cc?
        if(dsu.find(u) != dsu.find(v)){
            dsu.unite(u,v);
            res += w;
            aristas++;
        }
        if(aristas == n-1) break;
    }
}
```

```

    if(aristas == n-1) cout<<res<<'\\n';
    else cout<<"IMPOSSIBLE\\n";
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<tuple<int,int,int>> E(m); //(costo,u,v)
    // lleno la lista de aristas y llamo a kruskal
    kruskal(E,n);

    return 0;
}

```

Para el parcial

- Según si el grafo es denso o raro conviene usar Prim o Kruskal. Aún así para los parciales pueden asumir que tienen un algoritmo mágico que resuelve AGM en $O(\min(m * \log(n), n^2))$. También existen versiones de Prim y Kruskal para grafos raros/densos

Algoritmo de tarjan para puntos de articulación (o puentes)

Dado un grafo $G = (V, E)$, un **punto de articulación** “v”, es un vertice que tal que si lo removemos junto con sus aristas conectadas a él, el grafo se desconecta.

- Solución por fuerza bruta: ir sacando de a un vertice (y luego volver a ponerlo) y comprobar con DFS o BSF si el grafo siendo conexo. Costo $O(n * (n + m))$

Algoritmo de tarjan para puntos de articulación

Se basa en DFS por lo que tiene costo $O(n + m)$. El algoritmo usa 4 arreglos: - **visitado**: para tener un seguimiento de los vertices visitados - **momento_descubrimiento**: momento en el DFS llegó por primera vez al vertice (se tiene una variable global *tiempo* que por cada vertice visitado se incrementa en 1 y empieza en 0) - **mínimo_descubrimiento_alcanzable**: guarda para cada vertice, el mínimo tiempo de descubrimiento que se puede alcanzar desde el subarbol enraizado desde el vertice (tanto por tree edges como por backedges). - **padre**: para guardar cual es el padre de cada vertice en el recorrido. La raíz tiene padre “-1”

Se inicializan los 3 con largo igual a n y todos en -1. Se empieza el DFS desde un vertice cualquiera y por cada vertice “u” alcanzado:

- Se marca u como visitado, se setea su momento de descubrimiento y se aumenta la variable tiempo.
- Por cada vecino v de u :
 - si v no está visitado, setear el padre de v como u y expandir el dfs hacia v . Cuando vuelva de la recursión de v , setear el `minimo_momento_visita` de u como el $\min(\text{minimo_momento_visita}[u], \text{minimo_momento_visita}[v])$
 - Si u no es la raíz y $\text{minimo_descubrimiento_alcanzable}[v] \geq \text{momento_descubrimiento}[u]$ significa que desde el subarbol de v no hay ninguna arista que conecte con un ancestro de u (es decir, que tenga un momento de descubrimiento menor a u). Y por lo tanto u es un punto de articulación.
 - Si u era la raíz y tiene al menos 2 hijos, entonces es un punto de articulación.

```
from collections import defaultdict
```

```

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

```

```

def dfs(self, u, visited, discovery_time, low, parent, time, articulation_points):
    visited[u] = True
    discovery_time[u] = time
    low[u] = time # mínimo_descubrimiento_alcanzable
    children = 0

    for v in self.graph[u]:
        if not visited[v]:
            parent[v] = u
            children += 1
            self.dfs(v, visited, discovery_time, low, parent, time + 1, articulation_points)

            low[u] = min(low[u], low[v])

            if parent[u] == -1 and children > 1:
                articulation_points.add(u)
            elif parent[u] != -1 and low[v] >= discovery_time[u]:
                articulation_points.add(u)
        elif v != parent[u]:
            low[u] = min(low[u], discovery_time[v])

def find_articulation_points(self):
    visited = [False] * self.V
    discovery_time = [-1] * self.V
    low = [-1] * self.V
    parent = [-1] * self.V
    time = 0
    articulation_points = set()

    for u in range(self.V):
        if not visited[u]:
            self.dfs(u, visited, discovery_time, low, parent, time, articulation_points)

    return list(articulation_points)

```

Algoritmo de tarjan para puentes

Es una idea similar a la anterior, dado un vertice v en el recorrido DFS, por cada vecino u nos fijamos que no haya una backedge en el subarbol de u que vaya a un ancestro de v . O lo que sería que si el subarbol u tiene un *mínimo_descubrimiento_alcanzable* más grande que el momento de descubrimiento de v entonces la arista (v, u) es un puente. - Complejidad: $O(n + m)$

```

int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> tin, low; // momento_descubrimiento, mínimo_descubrimiento_alcanzable
int timer;

// p: padre (parent)
void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {

```

```

        dfs(to, v);
        low[v] = min(low[v], low[to]);
        if (low[to] > tin[v])
            IS_BRIDGE(v, to);
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}

```

Propiedades útiles sacadas de las guías 2 y 3

Guía 2

- **Ej 5:** Sean P y Q dos caminos distintos de un grafo G que unen un vértice v con otro w , G tiene un ciclo cuyas aristas pertenecen a P o Q .
- **Ej 6:** G un grafo conexo, todo par de caminos simples de longitud máxima de G tiene un vertice en común
- **Ej 10:** Sea G un grafo de n vértices. $G - v$ es bipartito para todo $v \in V(G)$ si y solo si G es bipartito o un ciclo impar
- **Ej 14:** Si todos los vértices de un digrafo D tienen grado de salida mayor a 0, entonces D tiene un ciclo. Un digrafo D es acíclico si y solo si D es trivial o D tiene un vértice con $d_{out}(v) = 0$ tal que $D - \{v\}$ es acíclico
- **Ej 15:** $O(\sum_{v \in V(G)} d(v)) = O(m)$. Todo grafo tiene $O(\sqrt{m})$ vértices con grado al menos \sqrt{m} .

Guía 3

- **Ej 2:** Una arista de un grafo G es puente si su remoción aumenta la cantidad de componentes conexas. Sea T un árbol DFS de un grafo conexo G , vw es un puente de G si y solo si vw no pertenece a ningún ciclo de G . Si $vw \in E(G) \cap E(T)$, entonces v es un ancestro de w en T o viceversa. Sea $vw \in E(G)$ una arista tal que el nivel de v en T es menor o igual al nivel de w en T , vw es puente si y solo si v es el padre de w en T y ninguna arista de $G - \{vw\}$ une a un descendiente de w (o a w) con un ancestro de v (o con v).
- **Ej 5:** Un árbol generador T de un grafo G es v -geodésico si la distancia entre v y w en T es igual a la distancia entre v y w en G para todo $w \in V(G)$. Todo árbol BFS de G enraizado en v es v -geodésico. La vuelta no vale, no todo árbol generador v -geodésico de un grafo G pueda ser generado con BFS sobre G desde v .
- **Ej 13:** Sea G un digrafo, si los pesos de G son todos distintos, entonces G tiene un único árbol generador mínimo.

Camino mínimo para grafos con peso y dirección

Dado un grafo $G = (V, E)$ y su función de pesos $w : E \rightarrow \mathbb{R}$, el **peso de un camino** $w(p)$ para $p = (v_0, \dots, v_k)$ es la suma de los pesos de cada arista: $\sum_{i=1}^k w(v_{i-1}, v_i)$. Definimos el camino más corto de v a u como

$\delta(u, v) = \min\{w(p) : u \rightarrow^p v\}$ si existe un camino de u a v
 $\delta(u, v) = \infty$ en caso contrario

Además si el grafo tiene ciclos negativos (por tener aristas con costo negativo) tal que un camino de v a u contiene un ciclo negativo, entonces el camino más corto de v a u no se puede definir y decimos que tiene costo $-\infty$.

También podemos decir que el camino más corto entre u y v no tiene ciclos de costo positivo (o 0), porque si los tuviera los podríamos sacar y obtendríamos un camino con un costo menor o igual. Vamos a asumir que cuando se dice caminos más cortos son caminos simples (sin ciclos).

Lema 22.1 (cormen)

Dado un grafo G dirigido y con peso, los subcaminos de un camino mínimo entre un vertice v y u también son caminos mínimos.

Algoritmos de camino mínimo para grafos con peso y dirección

Los algoritmos producen tanto el mínimo costo para llegar desde un vertice s a cualquier vertice alcanzable $v \in V$ cómo el camino en sí. Y como no tiene ciclos, es un árbol. Se lo llama *árbol de camino mínimo*.

Cada vertice guarda a su predecesor $v.\pi$ y un estimado del costo del camino mínimo $v.d$. Todos los algoritmos inicializan a $s.d = 0, s.\pi = NIL, v.\pi = NIL, v.d = \infty$ para todo $v \in V - \{s\}$.

Luego iterativamente se va recorriendo las aristas “relajandolas”, esto es chequear para una arista vu si $v.d > v.u + w(u, v)$. Si esto se cumple el camino de s a u y la arista uv es más corto que el camino más corto de s a v computado hasta ahora. Se actualiza $v.d$ y $v.\pi$.

Una vez terminado, cada vertice del árbol tiene el mínimo costo y si se va iterando por los predecesores, se puede obtener.

Propiedades del camino más corto y relajación

Triangle inequality (Lemma 22.10)

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 22.11)

We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 22.12)

If there is no path from s to v , then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 22.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v) , then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 22.15)

If $p = \langle v_0, v_1, \dots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k , and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p .

Predecessor-subgraph property (Lemma 22.17)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s .

Algoritmo de Bellman-Ford, Single-Source (con ciclos)

- **Complejidad:** $O(V^2 + VE)$
- Dado un grafo G dirigido y con peso, y una arista fuente “s”, si G no tiene ciclos negativos el algoritmo devuelve TRUE y un árbol de costo mínimo tal que cada vertice alcanzable desde “s” guarda su padre y el costo mínimo de llegar a él desde “s”. Si hay un ciclo negativo, devuelve FALSE.

```

BELLMAN-FORD( $G, w, s$ )
  INITIALIZE-SINGLE-SOURCE( $G, s$ )
  for  $i = 1$  to  $|G.V| - 1$ 
    for each edge  $(u, v) \in G.E$ 
      RELAX( $u, v, w$ )
  for each edge  $(u, v) \in G.E$ 
    if  $v.d > u.d + w(u, v)$ 
      return FALSE

```

```

return TRUE

RELAX(u, v, w)
  if v.d > u.d + w(u, v):
    v.d = u.d + w(u, v)
    v.π = u

INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex v ∈ G.V
    v.d = ∞
    d.π = NIL
  s.d = 0

```

Single-Source (sin ciclos)

Si ponemos la restricción de que el grafo dirigido con peso G sea aciclico (DAG), podemos hacer un algoritmo de camino más corto con costo $\Theta(V + E)$ usando topological sort (que tiene ese mismo costo).

Al terminar el algoritmo produce el arbol de costo mínimo tal que cada vertice guarda el costo mínimo del camino hasta el source y su padre.

```

DAG-SHORTEST-PATHS(G, w, s)
  topological_sort(G)
  INITIALIZE-SINGLE-SOURCE(G, s)

  for each vertex u ∈ G.V, taken in topological sort:
    for each vertex v in G.ady[u]:
      RELAX(u, v, w)

```

Algoritmo de Dijkstra, Single-Source (sin pesos negativos)

El algoritmo mantiene un conjunto S de los vertices que ya tienen el camino más corto hacia s calculado. Iterativamente toma el vertice $u \in V - S$ con el mínimo camino más corto estimado hasta el momento. Se agrega u a S y se relajan todas las aristas que salen de u .

El algoritmo de Dijkstra ejecutado sobre un grafo G con aristas no negativas y source s produce un subgrafo de predecesores G_π que es un arbol de caminos más cortos enraizado en s .

- **Complejidad:** Depende de la implementación del min-heap.
 - Si es un arreglo (aprovechando que los vertices se numeran de 1 a $|V|-1$) donde en cada posición se guarda el $v.d$, Insert y Decrease-key cuestan $O(1)$ y Extract-Min $O(V)$. Se hacen $O(V)$ Extract-Min y $O(E)$ Inserts y Decrease-keys. Queda $\mathbf{O(V^2 + E)} = \mathbf{O(V^2)}$
 - Podemos usar un binary-heap. Extract-Min con costo $O(\lg V)$ y se hace $|V|$ veces; construir el heap cuesta $O(V)$ (heapify); Decrease-key tiene costo $O(\lg V)$ y se hace $|E|$ veces. Queda un costo de $\mathbf{O((V + E) \lg V)}$ que en general (para grafos con más aristas que vértices) queda $\mathbf{O(E \lg V)}$. Si se cumple que $E = o(V^2/\lg V)$ entonces éste es una implementación mejor.
 - Con un fibonacci heap queda $\mathbf{O(V \lg V + E)}$

```

DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S = ∅
  Q = ∅ # min-heap

  for each vertex u ∈ G.V
    Insert(Q, u)

  while Q ≠ ∅
    u = Extract-Min(Q)
    S = Union(S, {u})
    for each vertex v in G.adj[u]
      RELAX(u, v, w)

```

```

if relax decreased v.d
    Decrease-Key(Q, v, v.d)

```

Algoritmo de Floyd-Warshall para todos a todos (sin ciclos negativos)

Se basa en la representación de grafos con matriz de adyacencia.

```

FLOYD-WARSHALL( $W, n$ )
1   $D^{(0)} = W$ 
2  for  $k = 1$  to  $n$ 
3      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
4      for  $i = 1$  to  $n$ 
5          for  $j = 1$  to  $n$ 
6               $d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$ 
7  return  $D^{(n)}$ 

```

- W : la matriz de costos
- D^0 : la matriz de costos mínimos para los caminos con ningún vértice intermedio.
- D^k : la matriz de costos mínimos para los caminos con $\{1, 2, \dots, k\}$ vértices intermedios.
- d_{ij}^k : representa el costo del camino mínimo de i a j con los $\{1, 2, \dots, k\}$ vértices como intermedios.
- n : la cantidad de vértices
- **Complejidad:** $\theta(V^3)$

Idea del algoritmo: Se numeran los vértices del grafo como $\{1, 2, \dots, n\}$. En cada paso k se busca el costo del camino mínimo que usa de intermedios a los vértices del subconjunto $\{1, 2, \dots, k\}$ de vértices. Con $k = 0$ corresponde a los caminos sin vértices intermedios, entonces solo tenemos los costos de las aristas que conectan directamente a i con j (es decir, los pesos de las aristas).

Luego para $k > 0$ vemos el mínimo costo de los caminos de i a j tal que se usen los vértices $\{1, \dots, k\}$ en dichos caminos. Puede pasar que k pertenezca al camino mínimo mejor que el computado hasta ahora sin k (con los vértices intermedios de $\{1, \dots, k-1\}$), por lo que el costo del camino mínimo desde i a k más el de k a j (ambos subcaminos no usan a k) es menor que el camino mínimo computado de i a j sin usar k de intermedio.

En síntesis la semántica de la recursión es: \mathbf{d}_{ij}^k el mínimo costo del camino de i a j usando los vértices $\{1, \dots, k\}$ es el mínimo entre \mathbf{d}_{ij}^{k-1} el mínimo costo del camino de i a j con los vértices $\{1, \dots, k-1\}$ de intermedios (sin usar a k), y $\mathbf{d}_{ik}^{k-1} + \mathbf{d}_{kj}^{k-1}$ el mínimo costo del camino de i a j usando de intermedio a los vértices $\{1, \dots, k-1\}$ y también a k (para calcular esto, vemos el mínimo camino desde i a k y desde k a j usando los vértices $\{1, \dots, k-1\}$).

Agregando la matriz de predecesores Se define la matriz de predecesores Π como aquella con elementos π_{ij} que contienen el predecesor de j para el camino mínimo de i a j . Si $i = j$ o no hay camino posible, $\pi_{ij} = \text{NIL}$.

Es decir, para recuperar el camino mínimo desde i hacia j hay que iterar sobre los antecesores de j empezando por π_{ij} hasta llegar a $\pi_{ii} = \text{NIL}$.

Se puede ir computando la matriz de predecesores $\Pi^0, \Pi^1, \dots, \Pi^k$ para los distintos k pasos del algoritmo de Floyd. ($k = n$ es el resultado del problema). Formulación recursiva para cada paso de k :

Para $k = 0$:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases}$$

Para $k > 0$:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \text{ (} k \text{ is an intermediate vertex)}, \\ \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \text{ (} k \text{ is not an intermediate vertex)}. \end{cases}$$

Si el camino usando k de intermedio es mejor, entonces hay que usar el predecesor dado por el camino de k a j .