Resumen Árbol Generador Mínimo

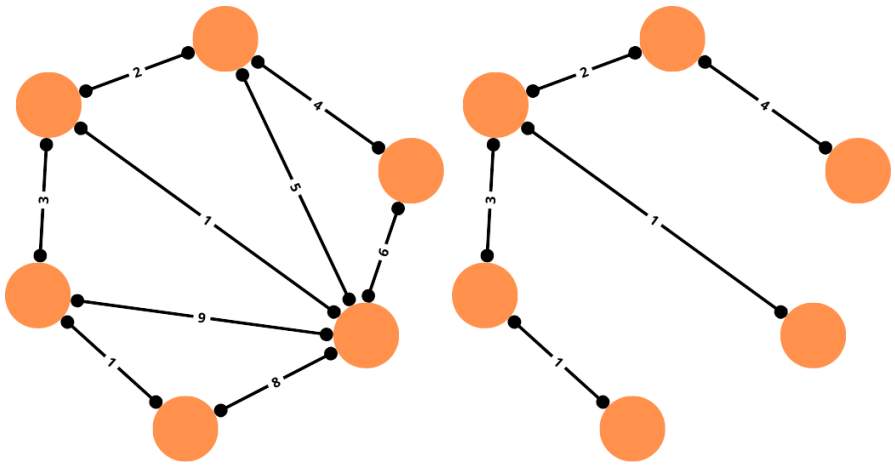
Tomás Felipe Melli

Noviembre 2024

Índice

1 Introducción

Imaginemos que tenemos que conectar $n - puntos$, pero cada conexión tiene un **costo** asociado. Podemos pensarlo como la distancia de ir de una provincia a otra $Jujuy \rightarrow Córdoba$ a través de cierta ruta. Supongamos ahora que queremos conectar *Salta* con el resto de las ciudades más importantes del país donde **todas las ciudades que están conectadas por estas rutas, minimizan el costo asociado a transitarlas para ir de una ciudad a otra.** Ahora, el problema es de **optimización combinatoria**. Si modelamos este problema con un grafo en el que los vértices son las ciudades y las aristas, las rutas, podemos definirles **el costo de transitarlas.** En escenarios como estos, en donde las aristas (puede suceder análogamente con los vértices o ambos al mismo tiempo) tiene un **costo asociado (también llamado peso)**, llamamos al grafo, **GRAFO PESADO.** Dicho esto, nos interesará encontrar el **ÁRBOL GENERADOR MÍNIMO** del grafo que modela el problema que queremos resolver. Es decir, un subgrafo generador del grafo que modela nuestro problema que es árbol. Por qué ? La respuesta es que queremos que sea **conexo** porque queremos que todas las ciudades estén conectadas entre sí y simultáneamente queremos minimizar la cantidad de enlaces (rutas en este caso) por ello $n - 1$ conexiones.
Miremos este ejemplo :



Sea $T = (V, X)$ un árbol y $l : X \rightarrow R$ una función que asigna costos a las aristas de T. Se define el costo de T como:

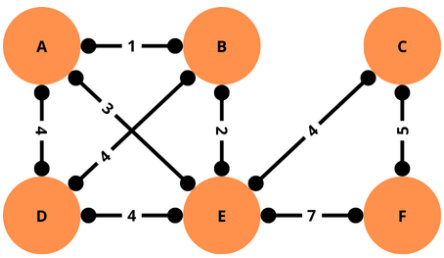
$$l(T) = \sum_{e \in T} l(e)$$

En este caso, tenemos definido un árbol donde sus aristas tienen pesos asociados en los que para saber el **costo de T** sumamos los pesos de sus aristas.

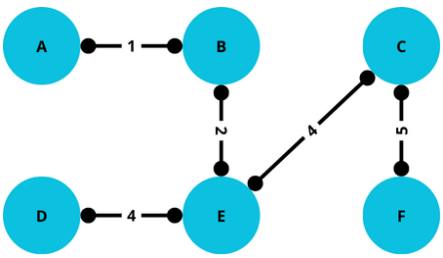
Dado un grafo $G = (V, X)$, un árbol generador mínimo de G, $\text{AGM}(G) = T$, es un árbol generador de G de mínimo costo, es decir:

$$l(T) \leq l(T') \quad \forall T' \text{ árbol generador de } G.$$

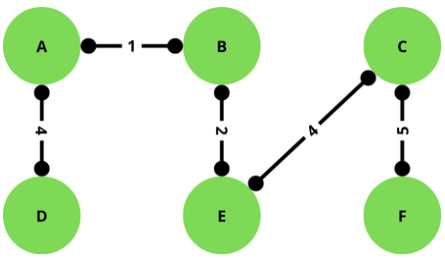
Si nosotros comparamos la sumatoria de los pesos de las aristas del AGM con otros árboles generadores, **el peso del AGM siempre será MENOR o IGUAL**. Por qué puede ser igual a otro árbol generador de G ? **No necesariamente es único ?**



(a) Sea G el grafo.



(b) Sea T un AGM de G de peso 16.



(c) Sea T' otro AGM de G de mismo peso

Ejemplo de un grafo G y dos árboles generadores mínimos.

Dado un grafo pesado en las aristas, $G = (V, X)$, el problema de árbol generador mínimo consiste en encontrar un AGM de G.

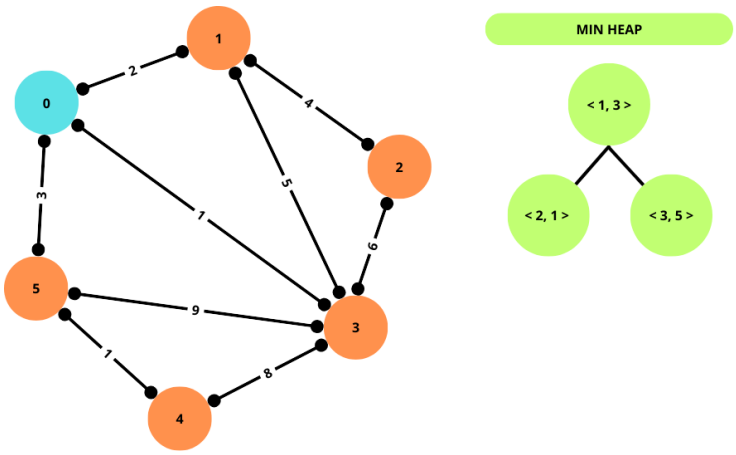
2 Algoritmos para hallar el Árbol Generador Mínimo de un Grafo Pesado

Los algortimos que vamos a ver en esta materia son : **Algoritmo de Prim** y **Algortimo de Kruskal**, ambos son golosos.

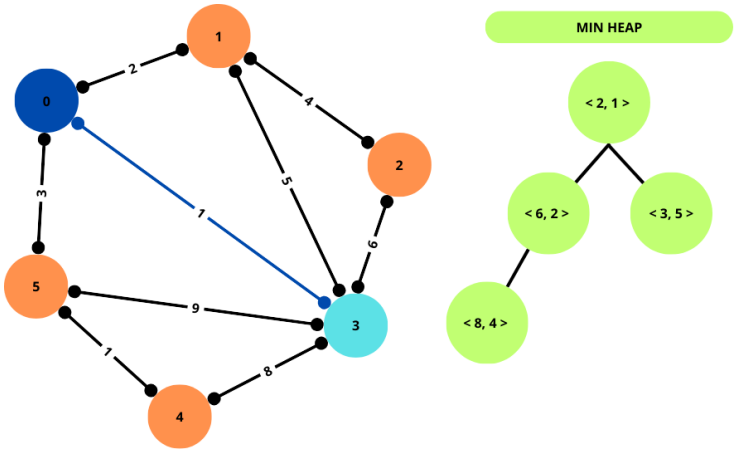
2.1 Algoritmo de Prim

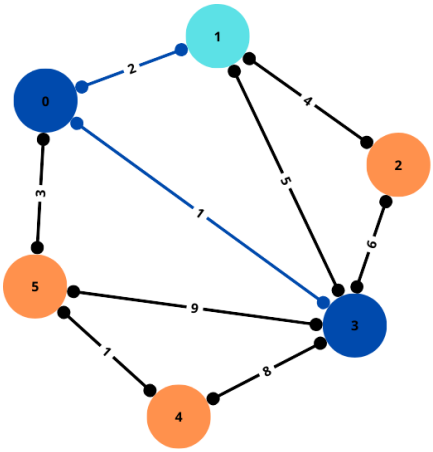
El algoritmo de Prim mantiene un árbol $T = (V_T, E_T)$, y en cada iteración agrega un nuevo vértice de $V - V_T$ al mismo (junto con una arista E_T), hasta que T pasa por todos los vértices de G (y se vuelve un árbol generador). La arista seleccionada en cada paso es la de mínimo peso entre todas las $(u, v) \in V_T \times (V - V_T)$ (las que conectan vértices dentro del árbol con los de afuera). El grafo **debe ser conexo**, eso asume este algoritmo. En conclusión, Prim enraiza en cierto vértice (asume que le grafo es conexo) y utiliza una estructura de datos (ya vamos a ver cuáles y qué complejidad nos permiten alcanzar) para almacenar los vértices $v \in V - V_T$ con el peso de la arista como **clave**. En esta exploración, (ya hablamos de la naturaleza greedy del algoritmo) tomará aquella **de menor peso** y lo agregará al árbol hasta que $V_T = V$.
En este ejemplo vamos a usar como estructura el min-heap:

- Enraizamos en 0. Creamos nuestro **MIN-HEAP** y pusheamos como **clave** el peso de cierta arista que une al nodo en exploración con vértices de $V - V_T$

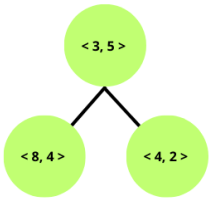


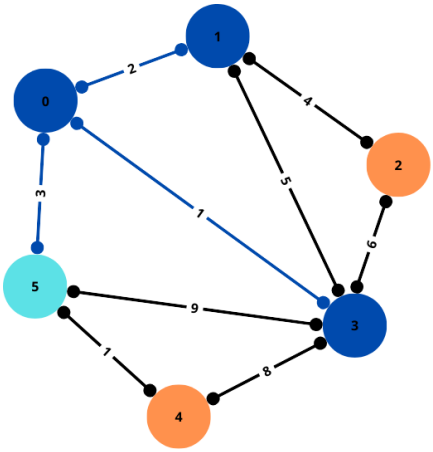
- Desencolamos $< 1, 3 >$ es decir, para llegar al vértice 3 tenemos una arista que nos cuesta 1, y la agregamos al árbol de Prim. Procedemos a explorar el vértice 3



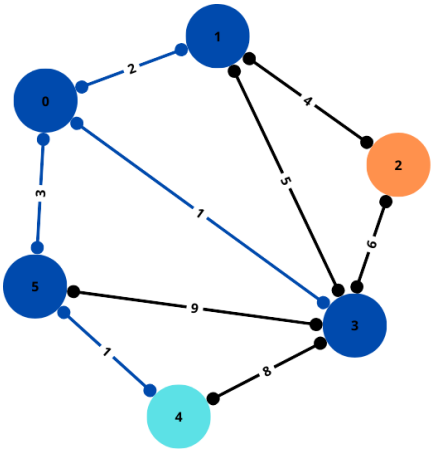


MIN HEAP

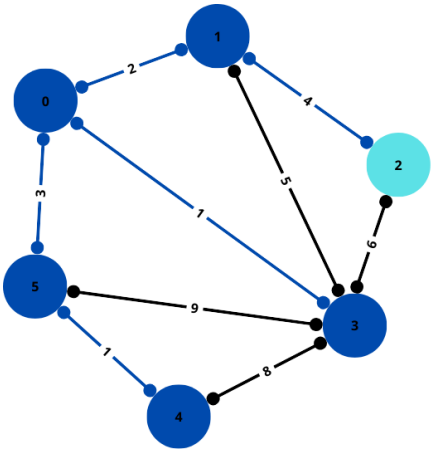




MIN HEAP

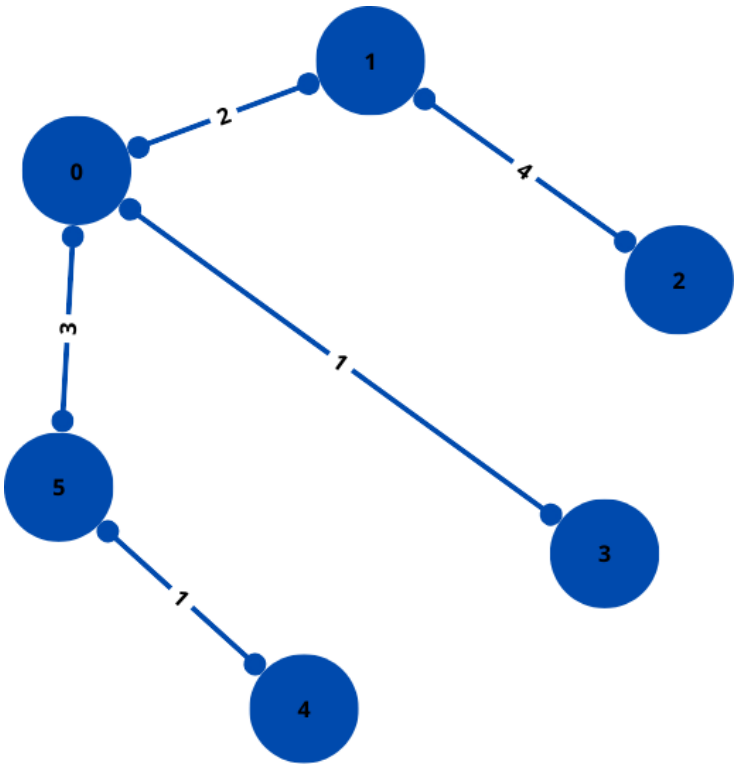


MIN HEAP



MIN HEAP

- Dijimos que la condición de corte del algoritmo es cuando $V_T = V$. El árbol que devuelve el algoritmo de Prim es un AGM:



2.1.1 Análisis de complejidad

Mencionamos al principio que dependiendo el uso de una estructura u otra la complejidad del algortimo se ve afectada.

a) **Array** : La idea es utilizar un arreglo de $|V|$ posiciones donde cada una indica el peso de l aarista que conecta cada vértice con el árbol (los vértices ya agregados tienen un valor particular). Para determinar cuál agregar al árbol en cierta iteración, se busca el mínimo, y se actualizan las posiciones que correspondan (dado que con este vértice nuevo tal vez se mejore el costo de llegar a algún otro). Esto tomaría $|V| - 1$ operaciones de complejidad $O(|V|)$ por tanto, tendríamos una complejidad de $O(|V|^2)$. Para grafos pesados la complejidad se mantiene.

b) Si utilizamos un **Min-Heap**, como el ejemplo que vimos, desencolamos la arista que nos conecta con un vértice $v \in V - V_T$ de costo mínimo (pues root del min-heap) y pusheamos los valores de las aristas que nos conectan a vértices de $V - V_T$. Esto quiere decir que tendremos, por iteración $deg(v) + 1$ operaciones logarítmicas (el grado del vértices pues sus adyacencias y 1 operación de desencolado). Esto nos da una complejidad de $O((|V| + |E|) \times log|V|)$. Dado que Prim asume G conexo donde $|E| \geq |V| - 1$, podemos concluir que a complejidad más ajustada es $O(|E|\ log|V|)$. La cosa cambia cuando el grafo es pesado, es decir, cuando $|E| \in \Omega(|V|^2)$, en este caso, se nos va un toque de mambo, $O(|V|^2\ log|V|)$. Para grafos ralos donde $|E| \in O(|V|)$, nos queda algo como $O(|V|\ log|V|)$.

c) La mejor manera es con **Fibonacci Heap**. El Fibonacci heap tiene tiempos amortizados mejores debido a su capacidad para realizar la operación `decrease-key` (es la que se usa para actualizar el valor de costo asociado al vértice $v \in V - V_T$) en tiempo constante en promedio y no tener que reestructurar todo el heap en cada operación. Con esto en mente, la complejidad ajustada para Prim implementado con FH es $O(|E| + |V|\ log|V|)$.

2.1.2 Código C++

```cpp
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};
int n;
vector<vector<Edge>> adj;
void prim()
{
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    vector<bool> selected(n, false);

    for (int i = 0; i < n; ++i)
    {
        if (q.empty()) {
            cout << "No hay AGM" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v])
        {
            if (!selected[e.to] && e.w < min_e[e.to].w)
            {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

2.2 Algoritmo de Kruskal

El algortimo de Kruskal tiene un approach diferente a Prim, si bien es greedy, la idea es **ordenar la aristas por su peso**. La idea de Kruskal es construir un bosque e ir iterando sobre las aristas previamente ordenadas crecientemente, y la manera de armar el AGM es **verificando si unen componentes conexas diferentes**. Ahora vamos a ver cómo lo logra y mediante el uso de cuáles estructuras.

2.2.1 Disjoint-Set

Esta estructura de datos lo que permite es almacenar **conjuntos disjuntos**. Es decir, nos permitirá mantener particiones (P_i) de cierto conjunto. La forma de lograr esto es, **asignando un referente** r_i **a cada** P_i. Tenemos estas operaciones definidas :

a) MAKE-SET(x): crea un nuevo conjunto dentro de la partición.

b) UNION(x,y): $x \in S_x$ e $y \in S_y$, esta función logra $S_x \cup S_y$.

c) FIND(x): Devuelve el representante de x.

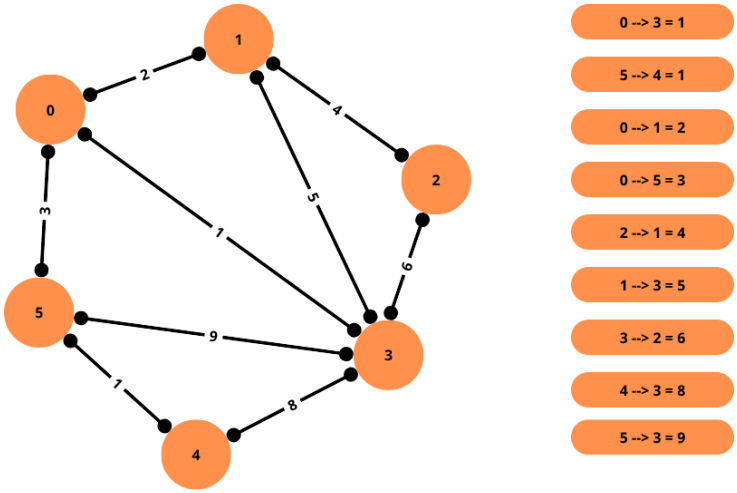
```
class DisjointSet {
    vector<int> rank, parent;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        for(int i = 0; i < n + 1; i++){
            parent[i] = i;
        }
    }

    int findSet(int node){
        // En caso que nodo sea el representante
        if (node == parent[node]) return node;

        // Hago path compression
        return parent[node] = findSet(parent[node]);
    }

    void unionByRank(int u, int v) {
        int uRepresentative = findSet(u);
        int vRepresentative = findSet(v);

        // Si tienen el mismo representante, entonces pertenece al
        // mismo conjunto
        if (uRepresentative == vRepresentative) return;

        // Actualizamos el representante segun el caso del rank
        if (rank[uRepresentative] < rank[vRepresentative]) {
            parent[uRepresentative] = vRepresentative;
        } else if(rank[uRepresentative] > rank[vRepresentative]) {
            parent[vRepresentative] = uRepresentative;
        } else {
            parent[vRepresentative] = uRepresentative;
            rank[uRepresentative]++;
        }
    }
};
```

En esta implementación se optimiza con **unión por rango** (es decir, a la hora de unir S_x con S_y se utiliza como raíz del nuevo árbol al representante de menor altura) y **path compression** (esto hace que para árboles profundos no tengamos que recorrer recursivamente padres para llegar al representante, simplemente hacemos que todos los nodos del espacio de búsqueda apunten al representante).

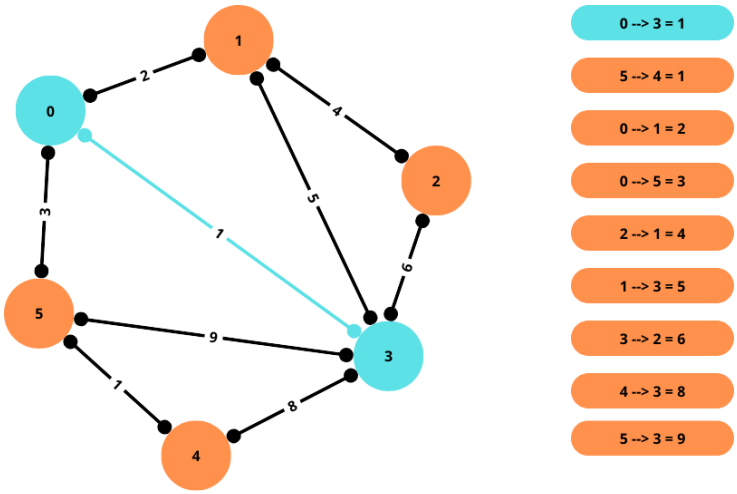
2.2.2 Visualización

Veamos cómo funciona :

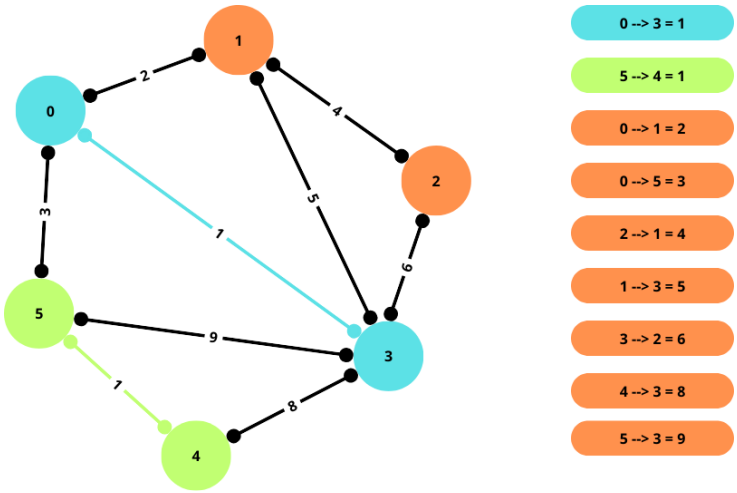
- Tenemos nuestro grafo G, le ordenamos **crecientemente** las aristas por su peso, creamos el Disjoint-Set para definir las particiones y ya estamos listos para comenzar. **Aclaración : las particiones en este momento inicial son $P_0 = \{0\}, P_1 = \{1\}, ..., P_n = \{n\}$.

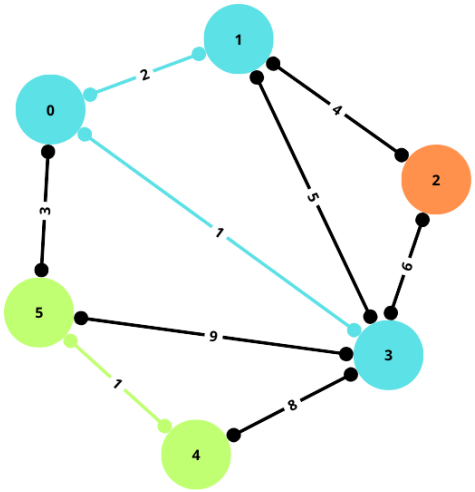


- Nos fijamos el representante de 0 y de 3 con la función $find()$. Como tienen representantes diferentes, entonces $union(0, 3)$.



- Seguimos la dinámica anterior, lo marcamos en distinto color por la partición.





0 --> 3 = 1

5 --> 4 = 1

0 --> 1 = 2

0 --> 5 = 3

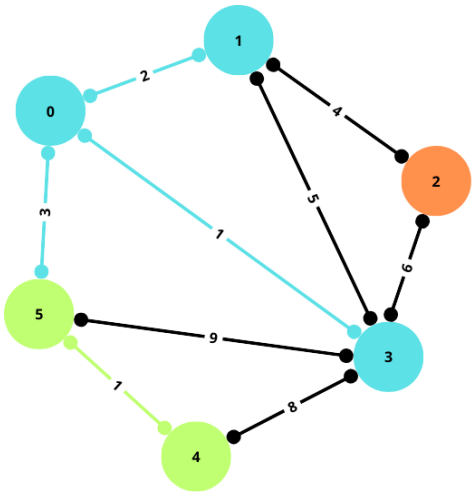
2 --> 1 = 4

1 --> 3 = 5

3 --> 2 = 6

4 --> 3 = 8

5 --> 3 = 9



0 --> 3 = 1

5 --> 4 = 1

0 --> 1 = 2

0 --> 5 = 3

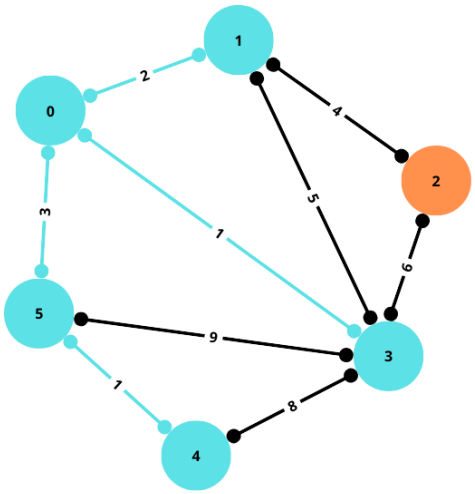
2 --> 1 = 4

1 --> 3 = 5

3 --> 2 = 6

4 --> 3 = 8

5 --> 3 = 9



0 --> 3 = 1

5 --> 4 = 1

0 --> 1 = 2

0 --> 5 = 3

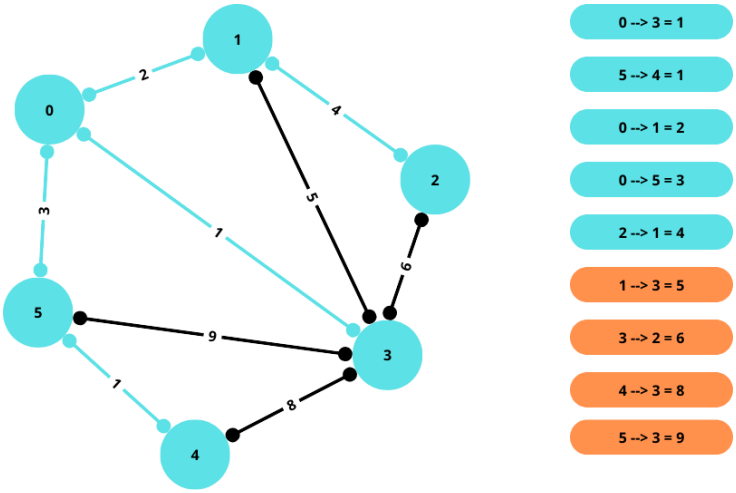
2 --> 1 = 4

1 --> 3 = 5

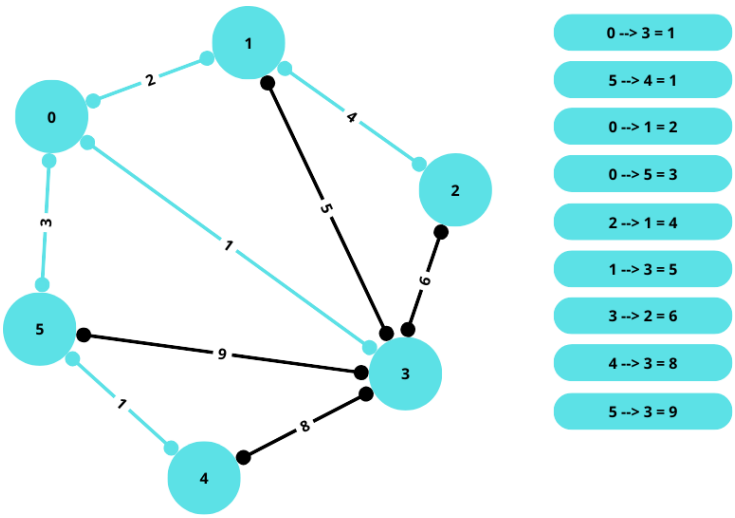
3 --> 2 = 6

4 --> 3 = 8

5 --> 3 = 9



- La condición de corte es : haber recorrido todas las aristas. Como desde $1 \rightarrow 3$ no conectan componentes conexas distintas, skipeamos hasta el final



2.2.3 Análisis de complejidad

Para poder dar una cota ajustada de la complejidad de Kruskal hay que pensar en lo siguiente : primero tenemos que ordenar crecientemente las aristas, esto toma $O(|E| \times log|E|)$. Luego dijimos que para cada arista tenemos que chequear cierta condición, es decir $|E| - veces$ llamamos a la función $union(S_x, S_y)$ que tiene complejidad $O(\alpha|V|)$. Resumiendo, tenemos

$$O(|E| \times log|V| + |E| \times \alpha|V|))$$

que podemos acotar por

$$O(|E| \times log|V|)$$

2.2.4 Código C++

```cpp
long long kruskal()
{
    int a, b;
    long long costo, costo_min = 0;

    // para cada arista (del conjunto de las aristas ordenadas ascendentemente por costo)
    for(int i = 0 ; i < aristas ; ++i)
    {
        a = arista[i].second.first;
        b = arista[i].second.second;
        costo = arista[i].first;

        // si est n en diferentes componentes conexas ==>
        if(encontrar_representante(a) != encontrar_representante(b))
        {
            // le sumo el costo
            costo_min += costo;
            // unimos componentes
            unir(a, b);
            // pusheamos la arista al AGM
            agm.push_back({costo, {a,b}});
        }
    }
    return costo_min;
}
```

3 Caminos

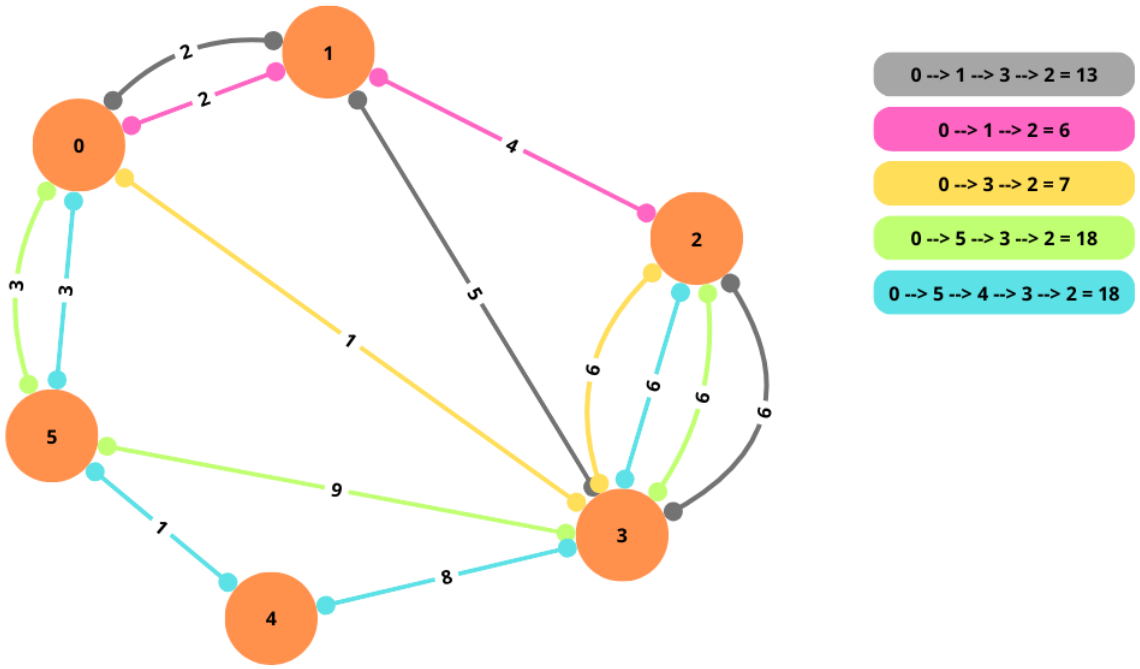
Hay problemas en los que nos interesará chequear cierta condición sobre la aristas de un camino entre dos nodos v_0, v_k. Podría ser que cierto recurso como el combustible para ir de una ciudad a otra esté limitado por cierta cantidad de kilómetros y por tal motivo, atravesar una ruta (arista) de costo mayor al que podemos tirar con la chata sea imposible y la idea será ver algún camino alternativo a esta. Mismo, podríamos querer transportar algo de un lado a otro y queremos encontrar el camino por el cuál podríamos mandar la mayor cantidad de ese recurso y aquella ruta que es la más restrictiva la querríamos evitar para poder mandar más recursos. Entre otros problemas comunes. Surge por ello, dos tipos de caminos que nos interesará investigar y manipular.

3.1 Camino Mini-Max

El **camino MINI-MAX** entre un par de vértices $u, v \in V$ de un grafo G donde tiene definida una función de peso para sus aristas $l : E \to N$, es aquel camino que **minimiza la arista más pesada del camino entre dicho par**. Es decir, el camino P_{min} cumple que :

$$P_{mini-max} = \arg\min \{\max \{l(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Miremos este grafo :



Miremos atentamente los distintos caminos que unen a $0 \to 2$. Si bien hay uno que minimiza el costo de ir desde $0 \to 2$, ahora queremos enfocar la atención en aquel que **minimiza la arista de mayor peso**, veamos :

- Camino Gris : $l(3 \to 2) = 6$.
- Camino Rosa : $l(1 \to 2) = 4$.
- Camino Amarillo : $l(3 \to 2) = 6$.
- Camino Verde : $l(5 \to 3) = 9$.
- Camino Celeste : $l(4 \to 3) = 8$.

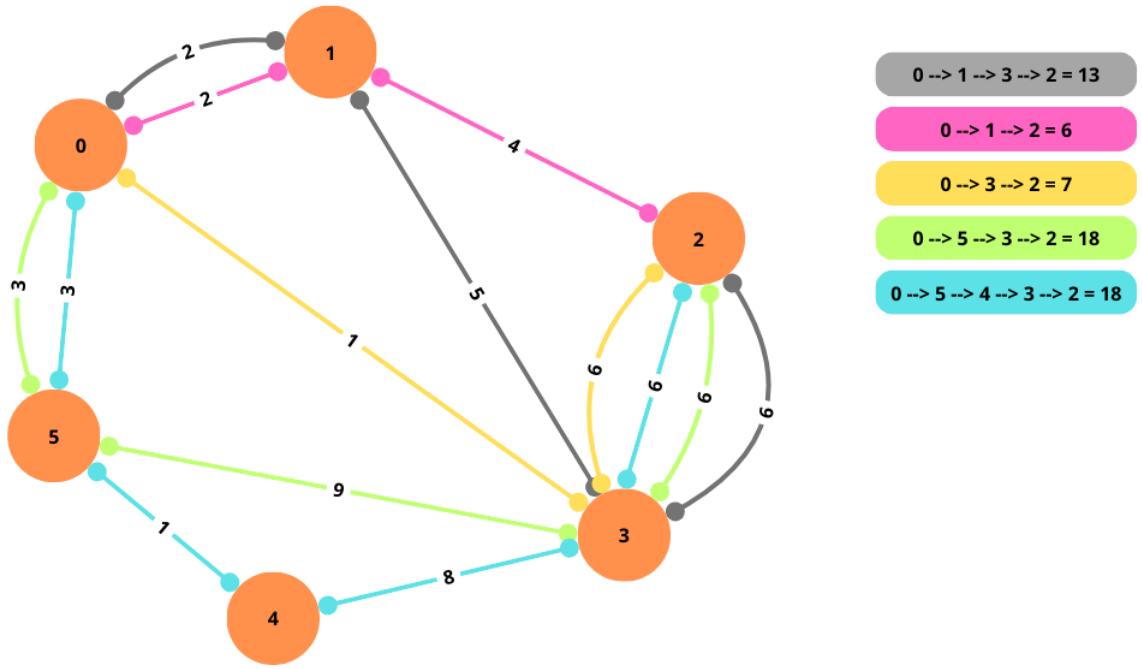
Dicho esto, aquél camino que minimiza la arista de mayor peso, es el camino Rosa.

3.2 Camino Maxi-Min

Por otro lado, tenemos bien definido el concepto de **camino MAXI-MIN** entre un par de vértices $u, v \in V$ de un grafo G donde tiene definida una función de peso para sus aristas $l : E \to N$, es aquel camino que **maximiza la arista más liviana del camino entre dicho par**. Es decir, el camino $P_{maxi-min}$ cumple que :

$$P_{maxi-min} = \arg\max \{\min \{l(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Miremos este grafo :



El camino que cumple que su arista más liviana dentro de las aristas más livianas de todos los otros caminos es la mayor, es $0 \to 5 \to 3$ (camino verde) donde $l(0 \to 3) = 3$ es mayor a las más livianas de los caminos :

- Gris : $l(0 \to 1) = 2$.
- Rosa : $l(0 \to 1) = 2$.
- Amarillo : $l(0 \to 3) = 1$.
- Celeste : $l(5 \to 4) = 1$.

4 Relación entre AGM y Caminos (Mini-Max y Maxi-Min)

Para encontrar un camino minimax entre dos vértices usamos el siguiente teorema:

Teorema. Dado un grafo pesado $G = (V, E)$ con $l : E \to N$ y un AGM T, para cualquier par de vértices $u, v \in V$, el camino que conecta a u con v en T es minimax.

Demostración. Supongamos que existe un camino mini-max $P_{mini-max}$ que pasa por aristas que no están en el árbol T. Sea $uv \in E \cap (P_{mini-max} - E_T)$ una arista que pertenece al grafo, al camino $P_{mini-max}$, pero no al conjunto de aristas del árbol, y T_{uv} el camino que conecta u con v en el árbol. Luego, si se toma $e' = \max\{l(e) \mid e \in T_{uv}\}$ como la arista de mayor peso de ese camino en el árbol que une a u con v , se puede ver que $l(uv) \geq l(e')$, ya que si no, e' podría ser reemplazada por uv, formando un AGM $T \cup \{uv\} - \{e'\}$ con un peso estrictamente menor, lo cual es absurdo porque T es AGM.

Por ende, cualquier arista fuera del árbol puede ser reemplazada por un camino que pasa por el árbol y tiene aristas de menor o igual peso (el peso máximo del camino se mantiene igual).

Esto significa que, para encontrar el camino mini-max entre cualquier par de vértices, se puede calcular el AGM del grafo, y tomar el camino que los une dentro de este.

Para el camino maximín, se puede tomar el árbol generador máximo. Se puede definir el árbol generador máximo, que maximiza el peso total de T si se toman los pesos $l'(e) = -l(e)$, es decir, le multiplicamos por -1 los costos de las aristas del grafo, y el peso de cualquier AG T será:

$$l'(T) = \sum_{e \in E_T} l'(e) = \sum_{e \in E_T} -l(e) = -l(T),$$

donde $l(T)$ es el peso del árbol generador mínimo. Como $l'(T) = -l(T)$, el problema de encontrar el árbol generador máximo es equivalente a resolver el problema del árbol generador mínimo utilizando los pesos negativos $l'(e)$.