







Grafos III : MST

Contenido

| | |
|-------------------------|---|
| 1. Conceptos básicos |  |
| 2. Algoritmo de Prim |  |
| 3. Algoritmo de Kruskal |  |
| 4. Propiedades |  |

Contenido

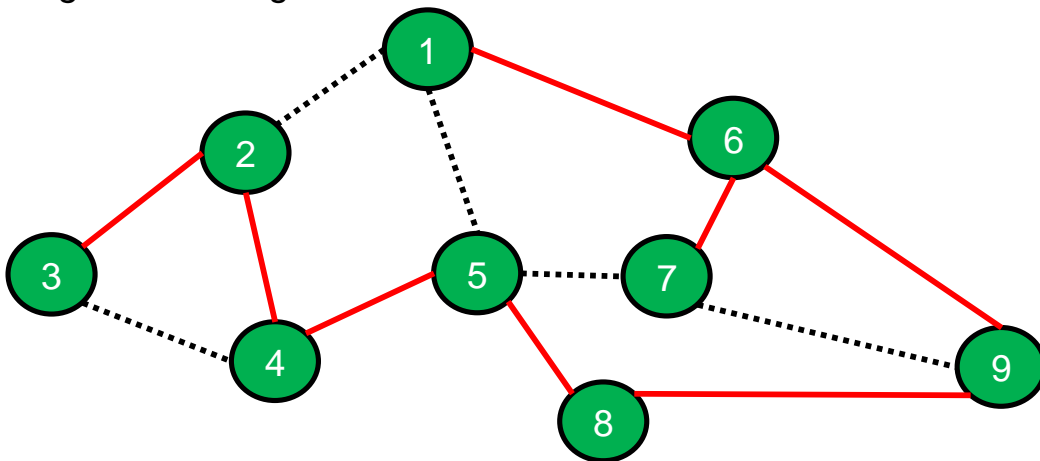
| | |
|-----------------------------|---|
| 1. Conceptos básicos |  |
| 2. Algoritmo de Prim |  |
| 3. Algoritmo de Kruskal |  |
| 4. Propiedades |  |

Spanning Tree

Definición: Un árbol de expansión (*spanning tree*) de un grafo no dirigido G es un subgrafo de G que contiene a todos los vértices de G y que sea conexo con la menor cantidad de aristas. Por lo tanto ese subgrafo tiene que ser un árbol.

Nota:

- Solo los grafos conexos tienen un *spanning tree*
- El *spanning tree* de un grafo conexo no necesariamente es único.



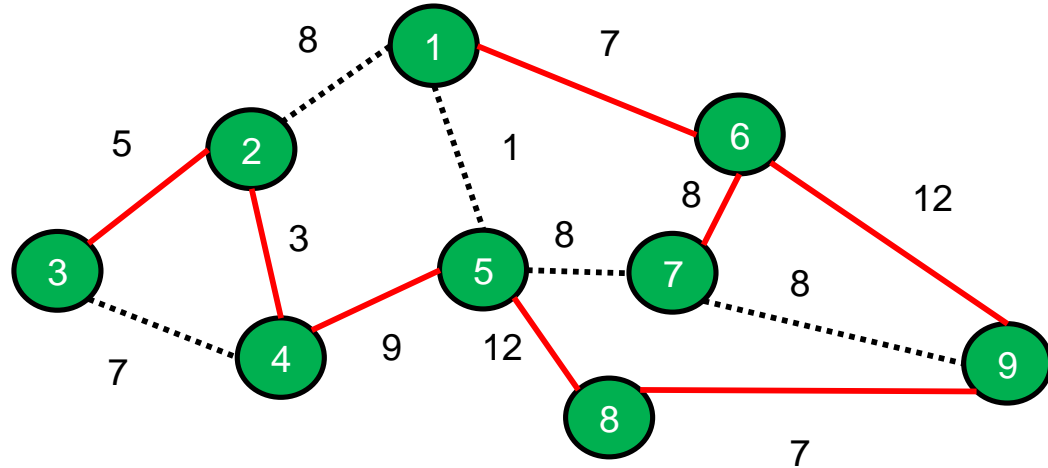
Peso de un Spanning Tree

Definición: Sea un *undirected weighted graph* G con el conjunto de vértices $V(G)$, con el conjunto de aristas $E(G)$ y con una función de pesos $w: E(G) \rightarrow \mathbb{R}$. Sea T uno de sus *spanning trees*. Definimos el peso total del *spanning tree* T como la suma de pesos de todas sus aristas.

Es decir $w(T) = \sum_{e \in E(T)} w(e)$

Ejemplo:

En la imagen de la derecha tenemos un Spanning Tree con peso $w(T) = 63$



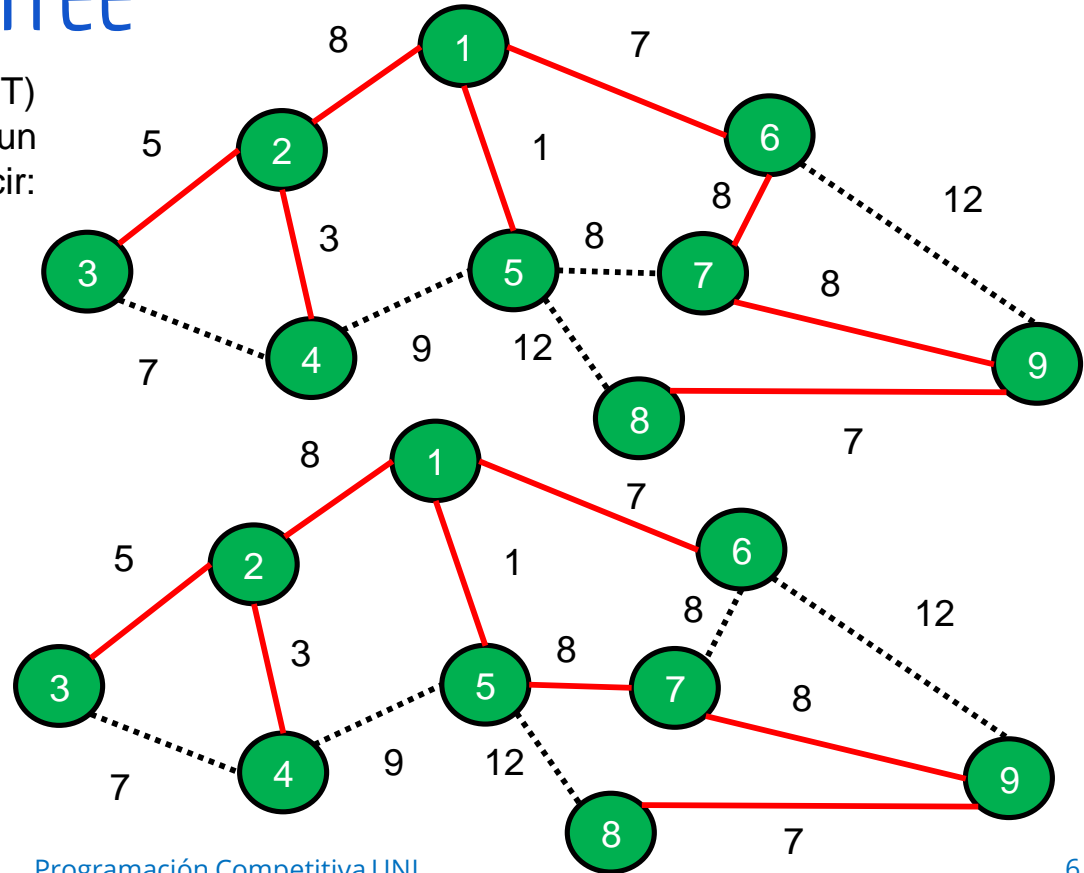
Minimum Spanning Tree

Definición: El *Minimum Spanning Tree* (MST) de un undirected weighted graph G es un Spanning Tree T con peso mínimo. Es decir: $w(T) \leq w(T'), \forall \text{ Spanning Tree } T'$

Nota:
El MST no necesariamente es único.

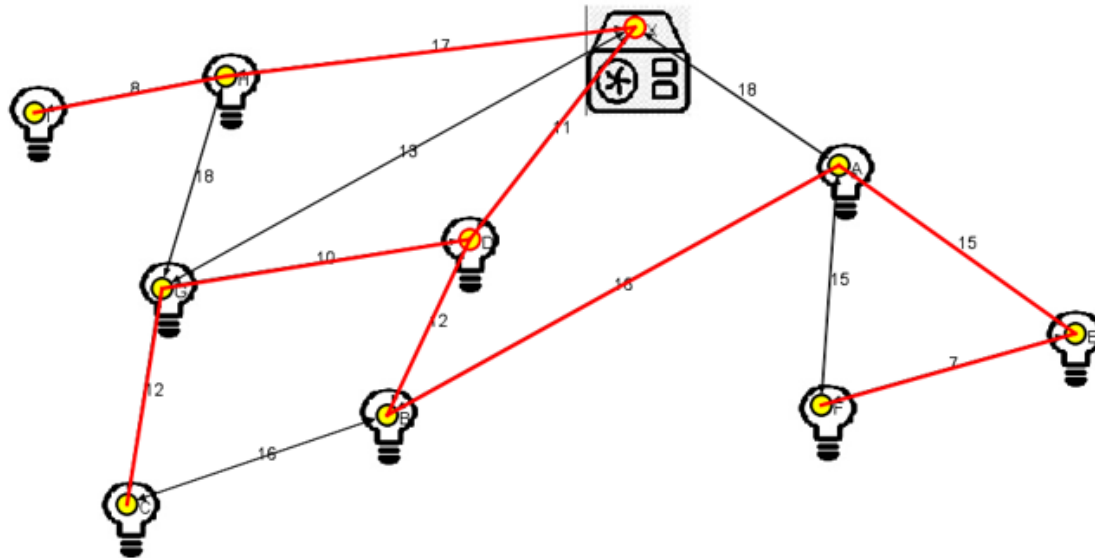
Ejemplo:

En la imagen de la derecha tenemos 2 Spanning Trees con peso $w(T) = 47$ y se puede demostrar que son de peso mínimo



Motivación

Encontrar el MST puede surgir de forma natural en problemas relacionados con redes. Por ejemplo, de suministrar energía a distintos puntos de una red y en donde cada arista representa un costo por la construcción de cierto cable. Se busca minimizar los costos.



Número de Spanning Trees

Antes de plantear los algoritmos para resolverlo, convendría analizar si el enfoque de **fuerzabruta** es factible. Es decir, ¿sería factible probar cada spanning tree y comparar cuál tiene el mínimo peso?





Para responder esta pregunta, debemos hacer un estimación de cuántos Spanning Tree puede tener un grafo.

Cayley's Formula: Un grafo **completo** de n nodos tiene exactamente n^{n-2} spanning trees.

Obviamente no todos los grafos son completos, pero esto nos da una idea de la cantidad exponencial de spanning trees que puede tener un grafo, por lo tanto el enfoque de fuerzabruta solo podría ser factible para n muy pequeños.

También existe una fórmula para obtener la cantidad de spanning trees en un grafo cualquiera utilizando el **Teorema de Kirchhoff** (<https://cp-algorithms.com/graph/kirchhoff-theorem.html>)

Contenido

| | |
|-----------------------------|---|
| 1. Conceptos básicos |  |
| 2. Algoritmo de Prim |  |
| 3. Algoritmo de Kruskal |  |
| 4. Propiedades |  |

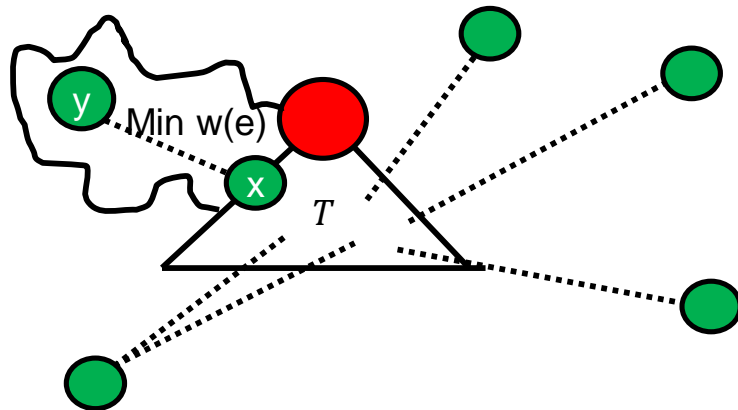
Algoritmo de Prim

Es un algoritmo **greedy** propuesto originalmente por el matemático Vojtěch Jarník en 1930, pero redescubierto y popularizado por Robert Clay Prim en 1957.

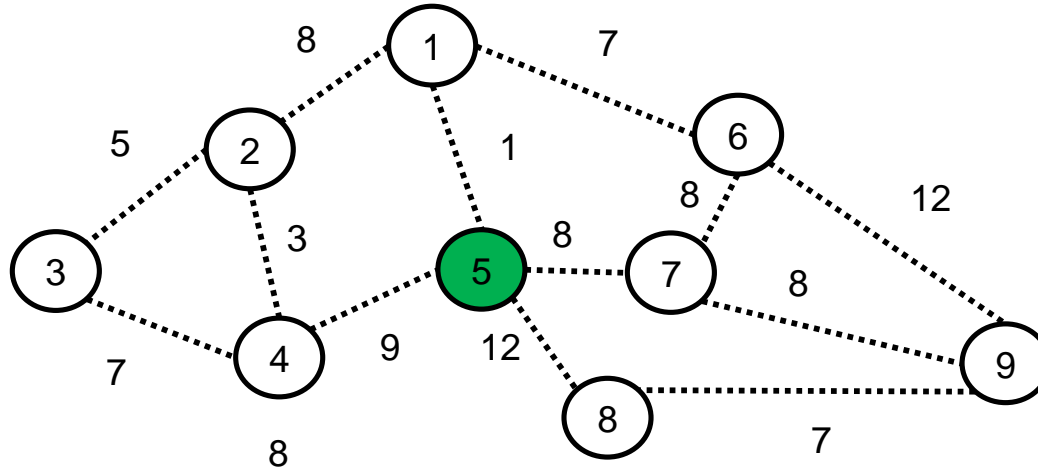
El algoritmo empieza con un árbol vacío T y selecciona cualquier nodo de G como raíz y lo añade al árbol. En cada iteración irá haciendo crecer el árbol, agregando de forma **greedy la arista de menor peso** que una a algún nodo del árbol T con algún otro que no esté en T . El algoritmo termina cuando T tenga exactamente n nodos (y sea un MST).

Algorithm 1: Prim's Algorithm

```
input :  $G(V_G, E_G, w)$   
output: Minimum Spanning Tree  $T(V_T, E_T, w)$ , Peso Total  
1  $T \leftarrow \emptyset$ ;  
2  $pesoTotal \leftarrow 0$ ;  
3 Escoge cualquier vértice  $r$  de  $V_G$ ;  
4  $V_T \leftarrow V_T \cup \{r\}$ ;  
5 while  $V_T \neq V_G$  do  
6   Encuentra la arista  $e = (x, y)$  tal que  $x \in V_T$ ,  $y \in V_G \setminus V_T$  y con  $w(e)$  mínimo;  
7    $V_T \leftarrow V_T \cup \{y\}$ ;  
8    $E_T \leftarrow E_T \cup \{e\}$ ;  
9    $pesoTotal \leftarrow pesoTotal + w(e)$ ;  
10 end  
11 return  $T(V_T, E_T, w)$ ,  $pesoTotal$ 
```



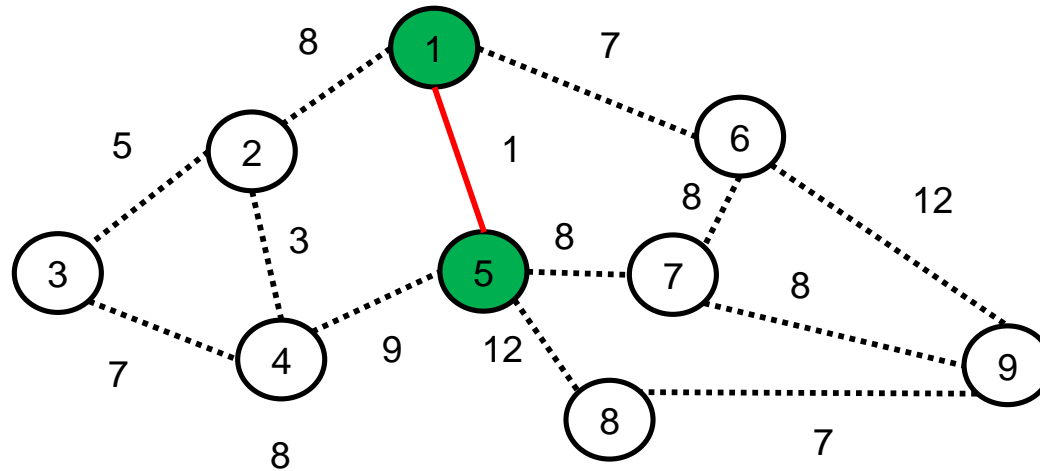
Ejemplo – Algoritmo de Prim



Escogemos el vértice 5

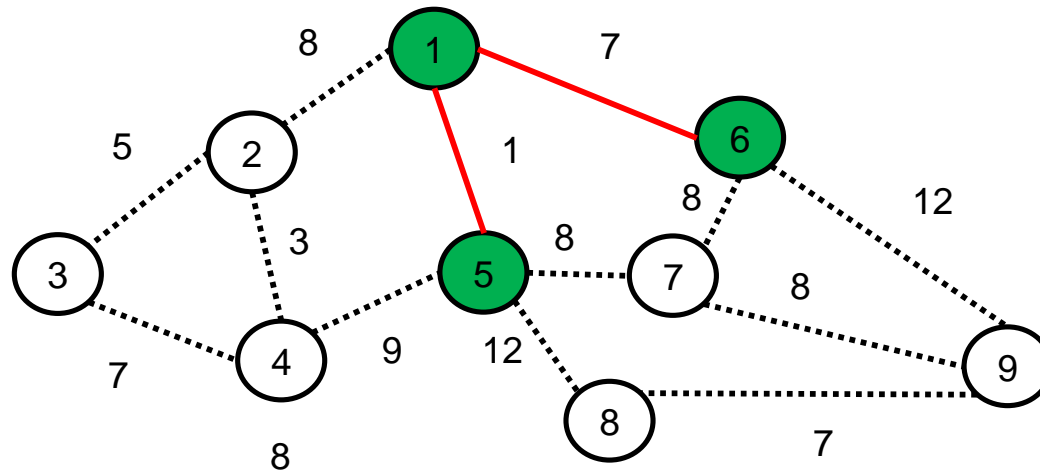
Peso total = 0

Ejemplo – Algoritmo de Prim



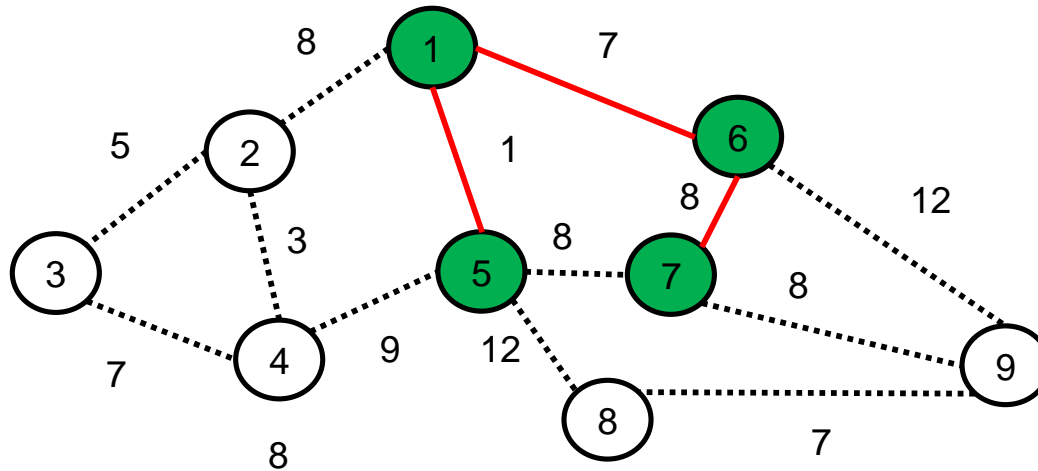
Peso total = 1

Ejemplo – Algoritmo de Prim



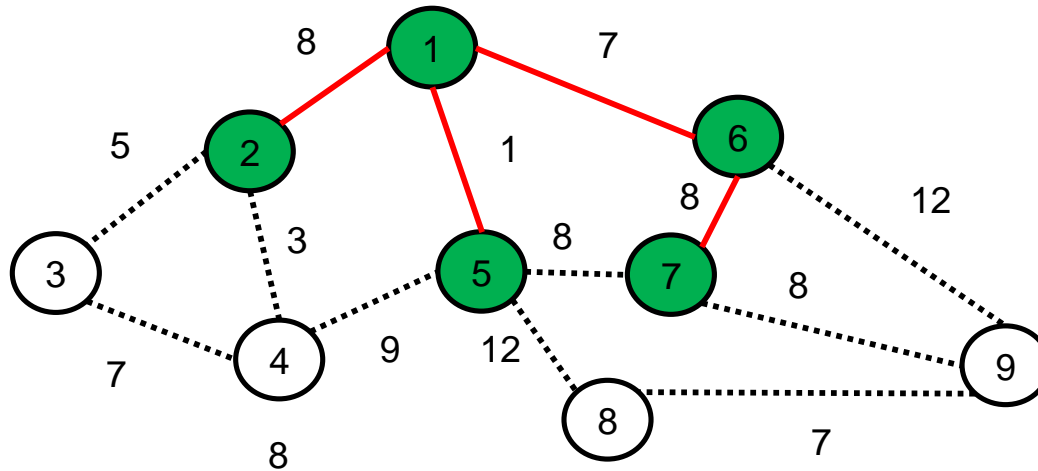
Peso total = 8

Ejemplo – Algoritmo de Prim



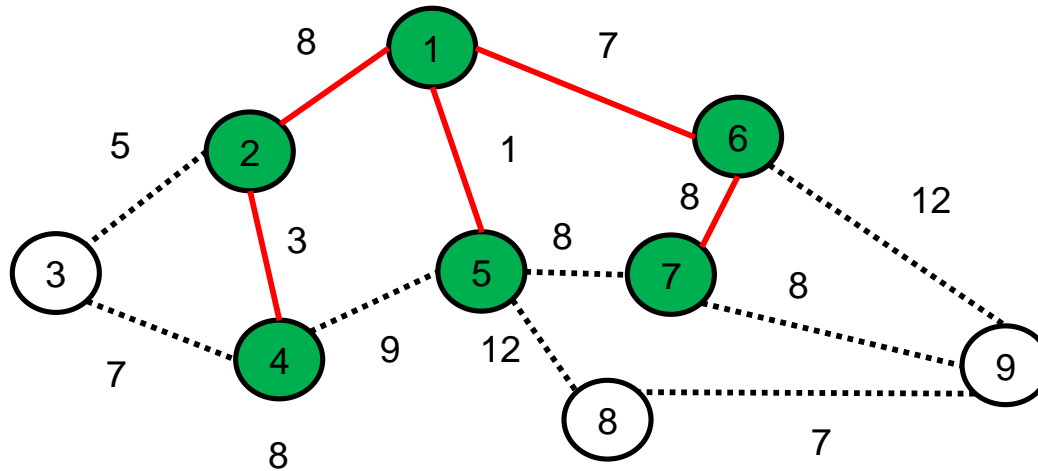
Peso total = 16

Ejemplo – Algoritmo de Prim



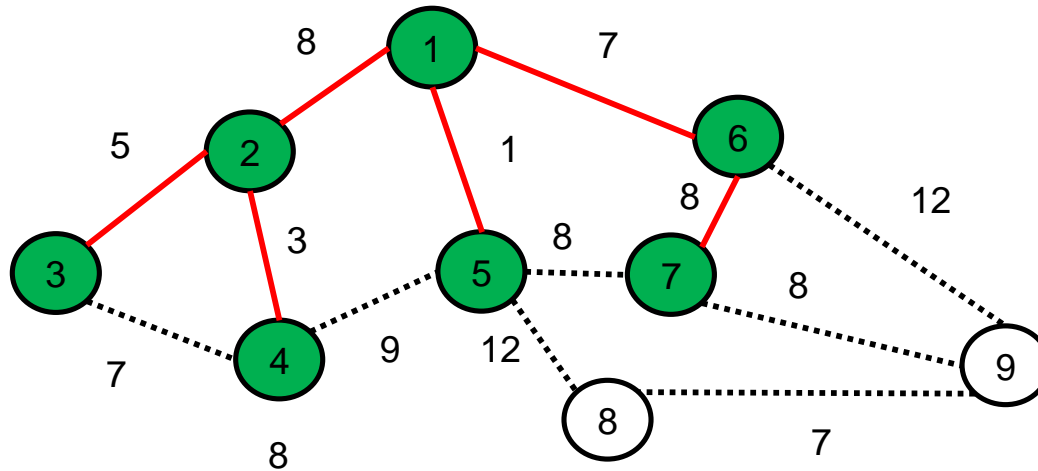
Peso total = 24

Ejemplo – Algoritmo de Prim



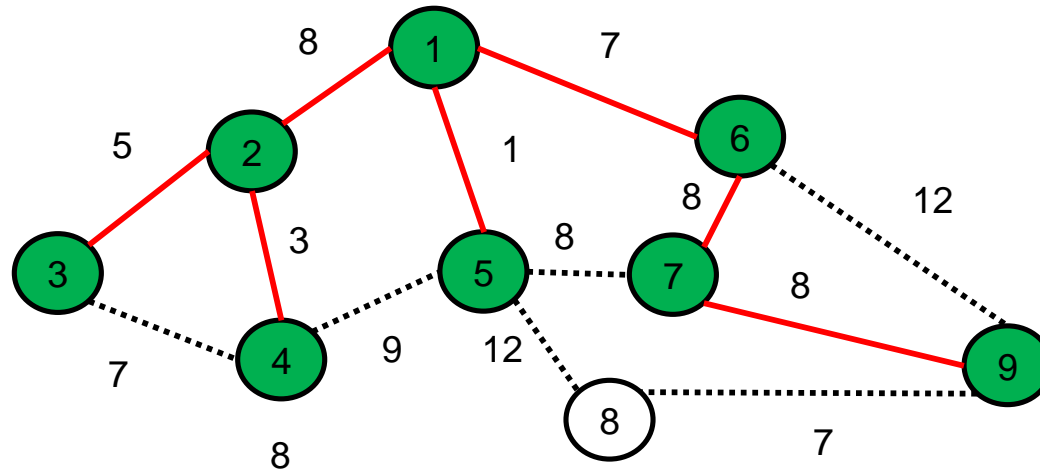
Peso total = 27

Ejemplo – Algoritmo de Prim



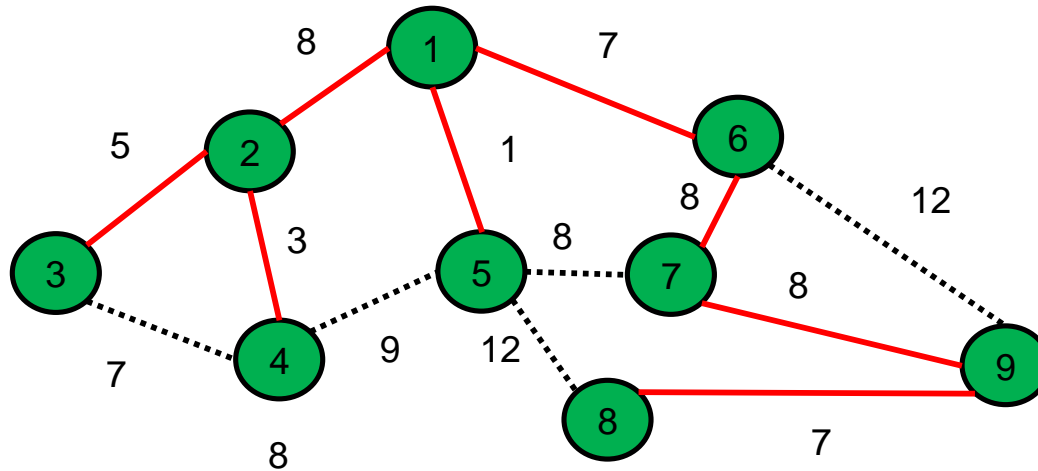
Peso total = 32

Ejemplo – Algoritmo de Prim



Peso total = 40

Ejemplo – Algoritmo de Prim



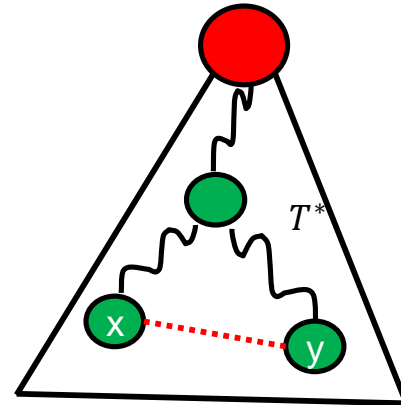
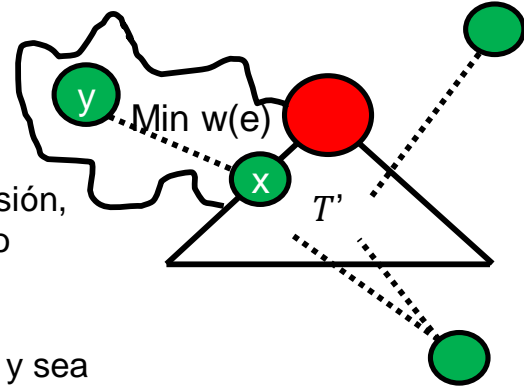
Peso total = 47

Proof of correctness

En primer lugar, es fácil notar que el grafo dado como respuesta es un árbol de expansión, debido a que en cada iteración añade una arista que no produce un ciclo y el algoritmo termina cuando se insertaron todos los vértices.

Para probar optimalidad, utilizaremos el **Exchange Argument**.

- 1) **Define las soluciones:** Sea T el árbol dado como output por nuestro algoritmo y sea T^* un MST
- 2) **Compara soluciones:** Supongamos que $T \neq T^*$, entonces habrá al menos una arista en donde varíen. Sea $e(x, y)$ la primera arista escogida por el algoritmo de Prim que no esté en T^* , y sea T' el árbol creado por Prim justo antes de agregar e . Sin pérdida de generalidad, supongamos que $x \in T'$, $y \notin T'$
- 3) **Intercambia elementos:** Si nosotros agregamos e a T^* produciríamos un ciclo. En ese ciclo, debe haber alguna arista $e'(u, v) \neq e$ con $u \in T'$, $v \notin T'$. Por lo tanto, e' también era candidata a ser elegida por Prim cuando se eligió e , así que se debe cumplir que $w(e) \leq w(e')$. Además, $T^* + e - e'$ seguirá siendo un árbol de expansión. De lo anterior se deduce que $w(T^* + e - e') \leq w(T^*)$. Pero como T^* era MST, entonces $T^* + e - e'$ también es MST
- 4) **Itera:** Cada vez que hacemos el procedimiento anterior producimos un MST que tiene una arista más en común con T . Si lo repetimos varias veces, conseguiremos un MST igual a T .



Implementación

Para la implementación utilizaremos un arreglo booleano $onTree[u]$ que nos diga si un determinado nodo u ya está en el MST o todavía.

Será necesario un arreglo $minWeight[u]$ que nos dirá, para un determinado nodo u que NO está en el árbol, cuál es el menor peso de la arista que lo lleva a un nodo del árbol. En caso el nodo u sí esté en el árbol. Nos dirá el peso que tiene con su padre en el MST.

Inicialmente $onTree[]$ deberá comenzar en **falso** para todos los nodos y $minWeight[]$ en $+\infty$.

En cada iteración del algoritmo debemos elegir el nodo que todavía no esté en el árbol ($onTree[u] = \text{false}$) y que tenga el peso mínimo hacia un nodo del árbol (mínimo $minWeight[u]$). Podemos hacer esa búsqueda en $O(n)$ y como hay n iteraciones, en total tendría una complejidad de $O(n^2)$

Implementación

```
const int MX = 1e5;  
const int INF = 1e9;
```

```
struct Endpoint{  
    int node , w;  
    Endpoint(int node, int w): node(node), w(w) {}  
};
```

```
vector<Endpoint> adj [MX];
```

```
int minWeight [MX];
```

```
int parent [MX];
```

Peso en el MST

Padre en el MST

```
void clear(int n) {  
    for (int i = 0; i < n; i++) {  
        adj[i].clear();  
    }  
}
```

```
void addEdge(int u, int v, int w) {  
    adj[u].push_back(Endpoint(v , w));  
    adj[v].push_back(Endpoint(u , w));  
}
```

```
int getMST(int n, int root = 0) { //O(n^2)  
    int totalWeight = 0;  
    vector<bool> onTree(n, false);  
    fill(minWeight, minWeight + n, INF);  
    minWeight[root] = 0;  
    parent[root] = -1;  
    for (int nodes = 1; nodes <= n; nodes++) {  
        int choice = -1;  
        for (int u = 0; u < n; u++) {  
            if (!onTree[u]) {  
                if (choice == -1 || minWeight[u] < minWeight[choice]) {  
                    choice = u;  
                }  
            }  
        }  
        if (minWeight[choice] == INF) {  
            //graph is not connected  
            return -1;  
        }  
        totalWeight += minWeight[choice];  
        onTree[choice] = true;  
        for (Endpoint e : adj[choice]) {  
            if (!onTree[e.node] && e.w < minWeight[e.node]) {  
                minWeight[e.node] = e.w;  
                parent[e.node] = e.w;  
            }  
        }  
    }  
    return totalWeight;  
}
```

Inicialización
de variables

Agregamos
nuevas
aristas
candidatas

Implementación eficiente

Una complejidad de $O(n^2)$ es buena para grafos densos en donde el número de aristas $m \approx \frac{n(n-1)}{2}$.

Sin embargo para grafos esparsos (*sparse graphs*) es muy lento. Fíjense que el cuello de botella está en buscar la arista de menor peso. Si lo hacemos con un data structure más potente como un *min heap* (*priority queue* en C++) podemos reducir esa búsqueda a una complejidad logarítmica. La cola de prioridad tendría un tamaño máximo de m (el número de aristas).

En cada iteración tendríamos una complejidad de $O(\log m)$ para obtener el mínimo y de $O(\deg(u) \times \log(m))$.

La complejidad total sería $O(n \log m + m \log m) = O(m \log m) = O(m \log n)$. El último paso se debe a que $m = O(n^2)$.

Detalle de implementación: En C++ el *priority_queue* es por defecto un max heap (nos retornaría la arista de peso máximo en vez del mínimo). Para utilizar un min heap deberemos declararlo así:

```
priority_queue<Endpoint, vector<Endpoint>, greater<Endpoint>> q;
```

Y deberemos definir el operador $>$ en Endpoint

Implementación eficiente

```
const Long MX = 1e5;
const Long INF = 1e18;

struct Endpoint{
    Long node , w;
    Endpoint(Long node, Long w): node(node), w(w) {}
    bool operator >(Endpoint const &other) const{
        return w > other.w;
    }
};

vector<Endpoint> adj [MX];
Long minWeight [MX];
Long parent [MX];

void clear(Long n) {
    for (int i = 0; i < n; i++) {
        adj[i].clear();
    }
}

void addEdge(Long u, Long v, Long w) {
    adj[u].push_back(Endpoint(v , w));
    adj[v].push_back(Endpoint(u , w));
}
```

Peso en el MST

Padre en el MST





```
Long getMST(Long n, Long root = 0) { //O(mlogn)
    Long totalWeight = 0;
    Long totalNodes = 0;
    vector<bool> onTree(n, false);
    fill(minWeight, minWeight + n, INF);
    minWeight[root] = 0;
    parent[root] = -1;
    priority_queue<Endpoint, vector<Endpoint>, greater<Endpoint>> q;
    q.push(Endpoint(root, 0));
    while (!q.empty()) {
        Endpoint cur = q.top();
        q.pop();
        Long u = cur.node;
        if (onTree[u]) {
            continue;
        }
        totalWeight += cur.w;
        onTree[u] = true;
        totalNodes++;
        for (Endpoint e : adj[u]) {
            Long v = e.node;
            if (e.w < minWeight[v] && !onTree[v]) {
                minWeight[v] = e.w;
                parent[v] = u;
                q.push(e);
            }
        }
    }
    if(totalNodes != n) return -1;
    return totalWeight;
}
```

Inicialización
de variables

Arista candidata
con peso mínimo

Agregamos
nuevas
aristas
candidatas

Contenido

| | |
|--------------------------------|---|
| 1. Conceptos básicos |  |
| 2. Algoritmo de Prim |  |
| 3. Algoritmo de Kruskal |  |
| 4. Propiedades |  |

Algoritmo de Kruskal

Es otro algoritmo **greedy** propuesto por el matemático Joseph Kruskal en 1956.

A diferencia de Prim, en vez de empezar con un árbol e irlo incrementando, en Kruskal se empieza con un **bosque** de n árboles compuestos por 1 solo nodo. En cada iteración se busca fusionar 2 árboles distintos trazando una arista entre ellos (al hacerlo NO se creará ningún ciclo y el grafo seguirá siendo un bosque). Para fusionar los árboles se busca **la arista de menor tamaño** que una 2 nodos que estén presentes en árboles distintos. Como en cada fusión el # de árboles disminuye en 1, luego de $n - 1$ fusiones se tendrá 1 solo árbol que será un árbol de expansión (y se demostrará que es mínimo).

Algorithm 2: Kruskal's Algorithm

input : $G(V_G, E_G, w)$

output: Minimum Spanning Tree $T(V_T, E_T, w)$, Peso Total

1 Ordena E_G en orden no decreciente según su peso;

2 $V_T \leftarrow V_G$;

3 $E_T \leftarrow \emptyset$;

4 $pesoTotal \leftarrow 0$;

5 **foreach** $e(x, y)$ en E_G **do**

6 **if** x está en distinto árbol que y **then**

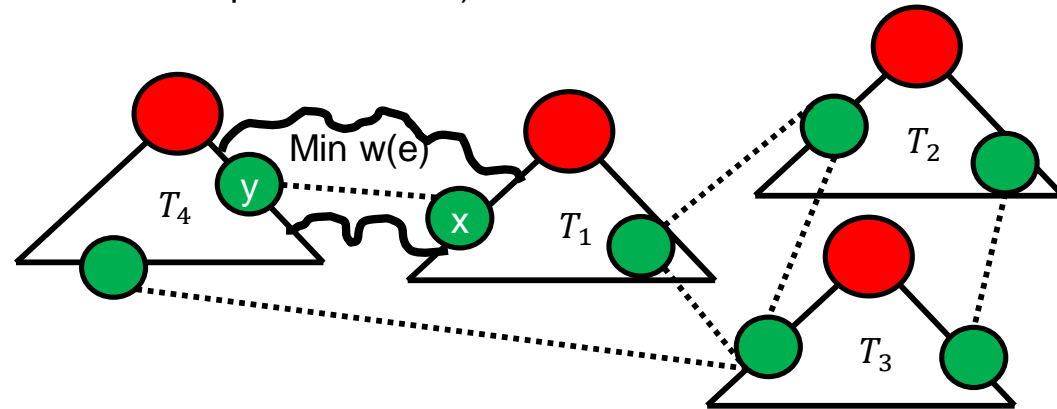
7 $E_T \leftarrow E_T \cup \{e\}$;

8 $pesoTotal \leftarrow pesoTotal + w(e)$;

9 **end**

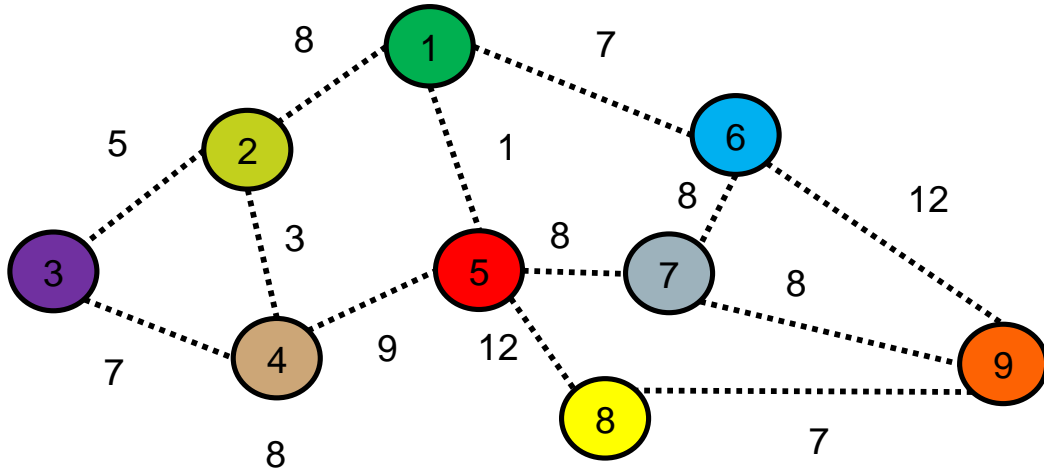
10 **end**

11 **return** $T(V_T, E_T, w)$, $pesoTotal$



Ejemplo – Algoritmo de Kruskal

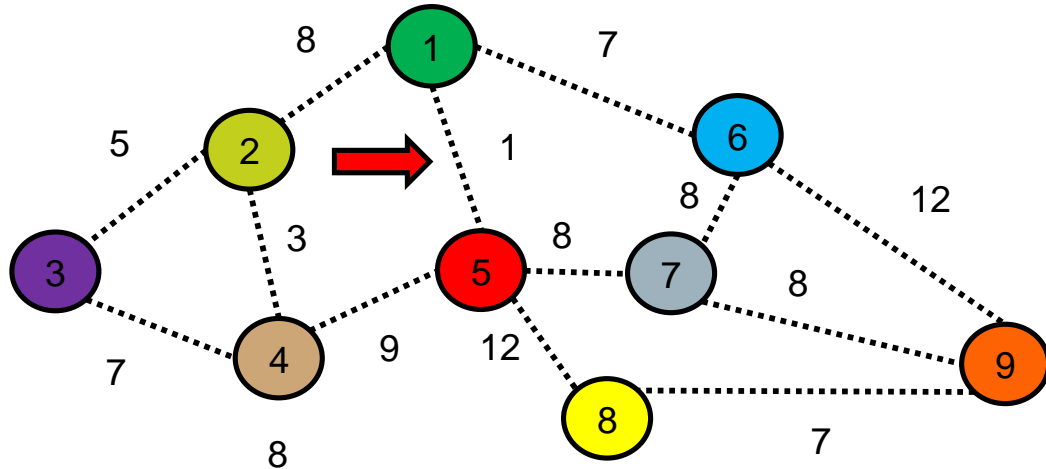
Peso total = 0



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |

Ejemplo – Algoritmo de Kruskal

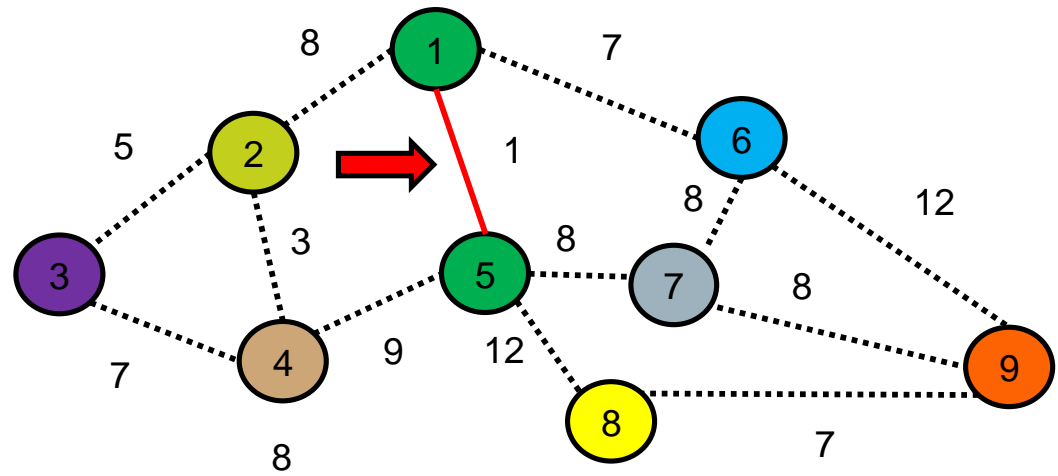
Peso total = 0



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |

Ejemplo – Algoritmo de Kruskal

Peso total = 1

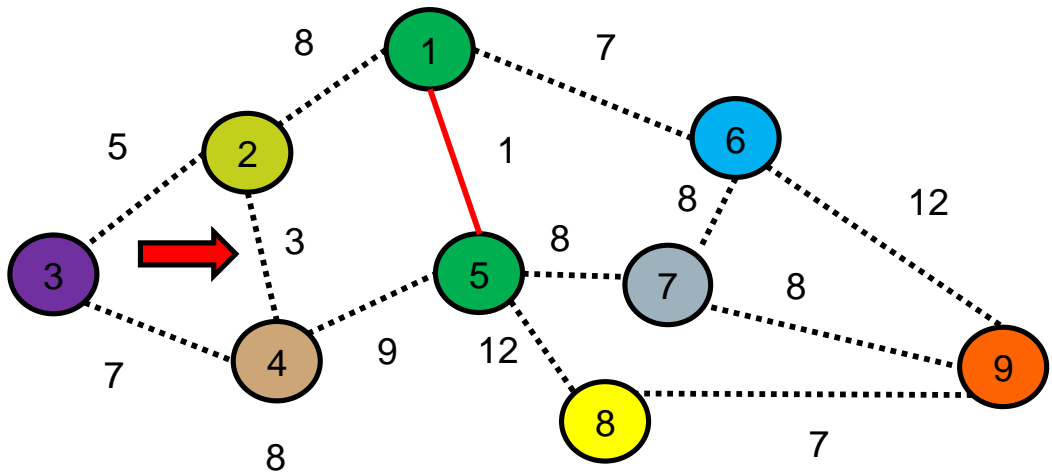


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 1

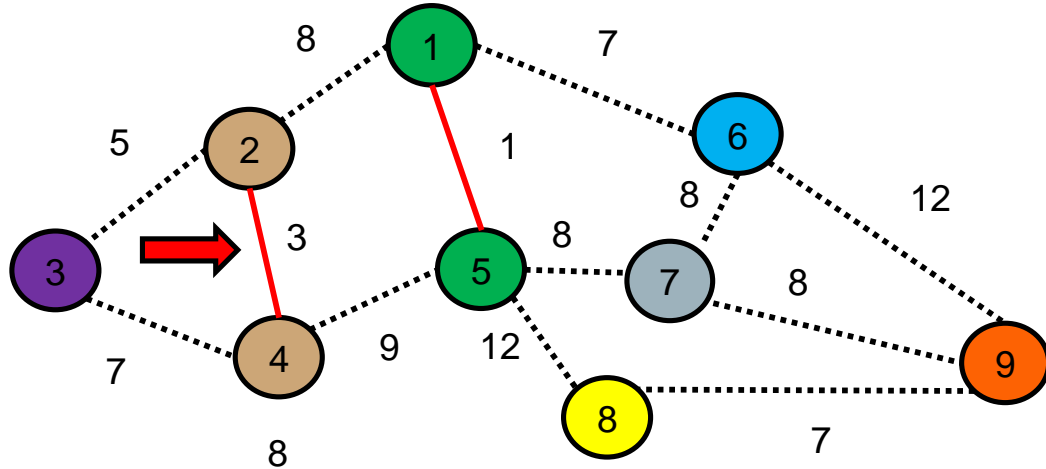


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 4

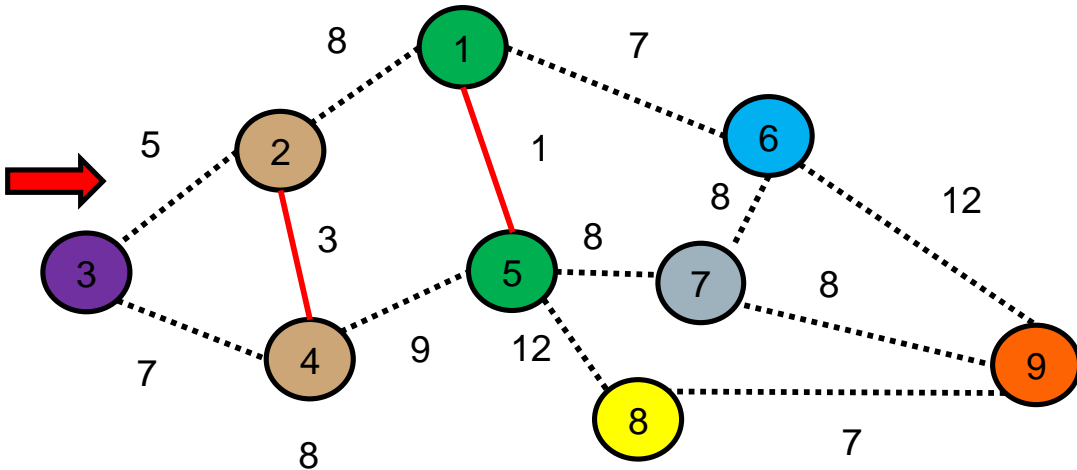


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 4

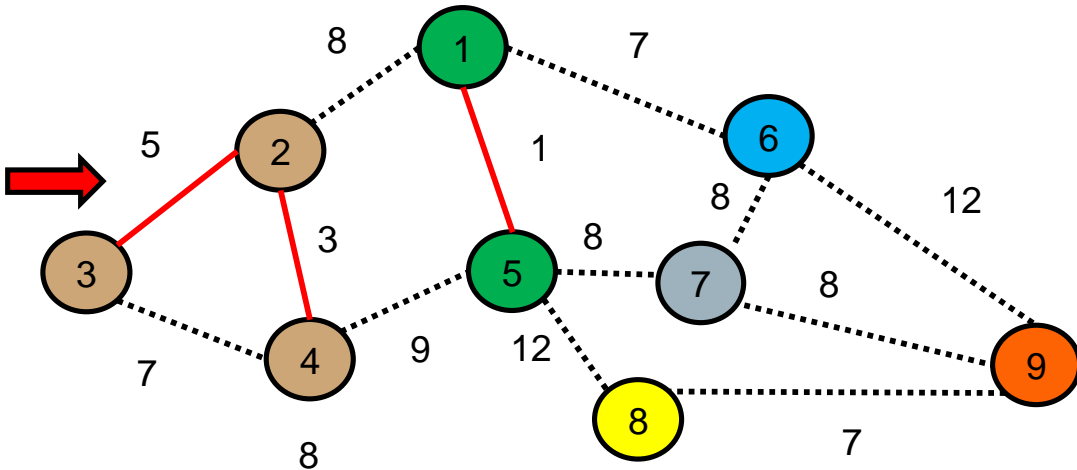


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

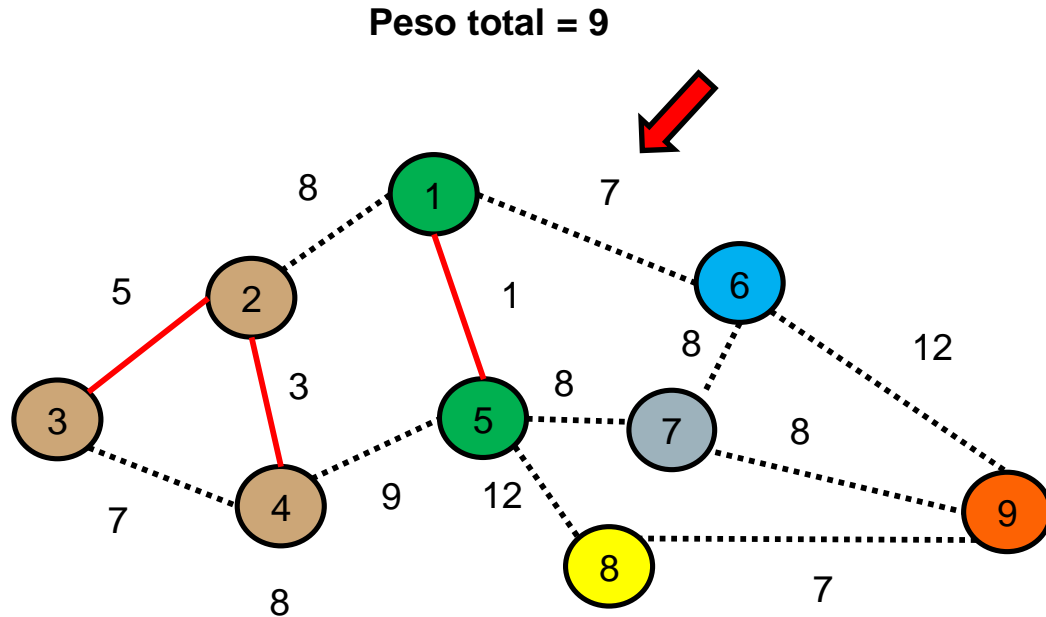
Peso total = 9



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



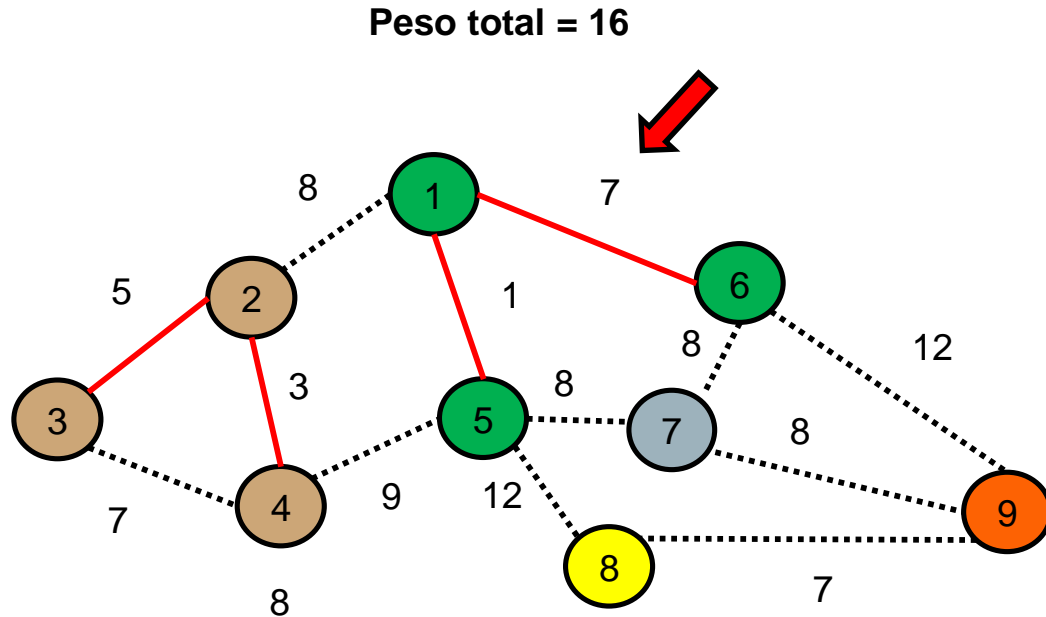
Ejemplo – Algoritmo de Kruskal



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

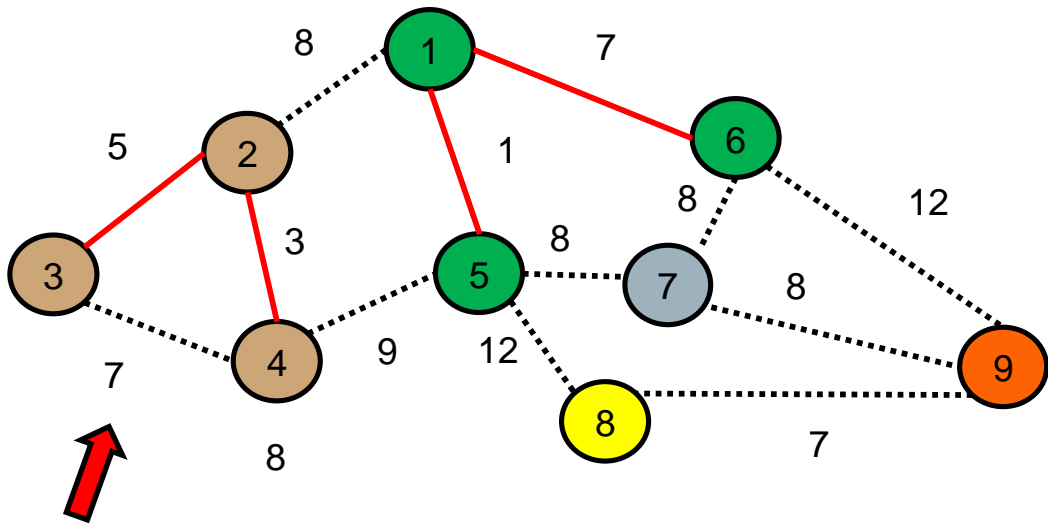


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 16

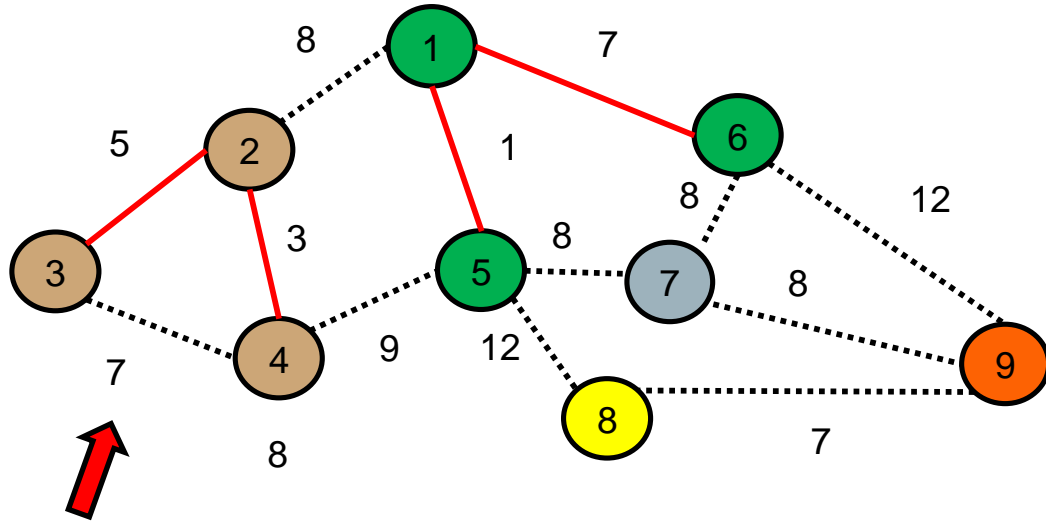


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 16

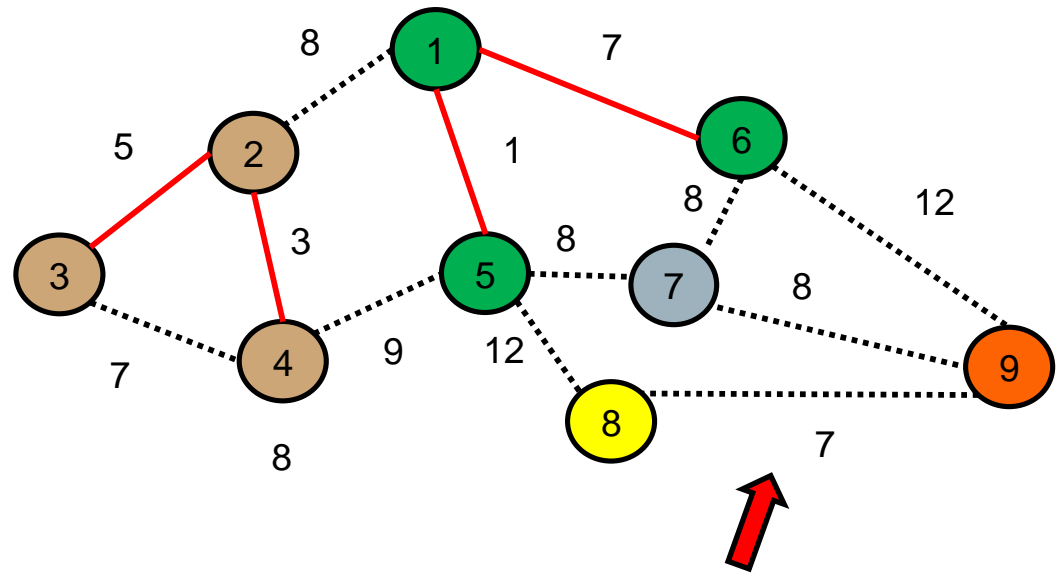


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 16

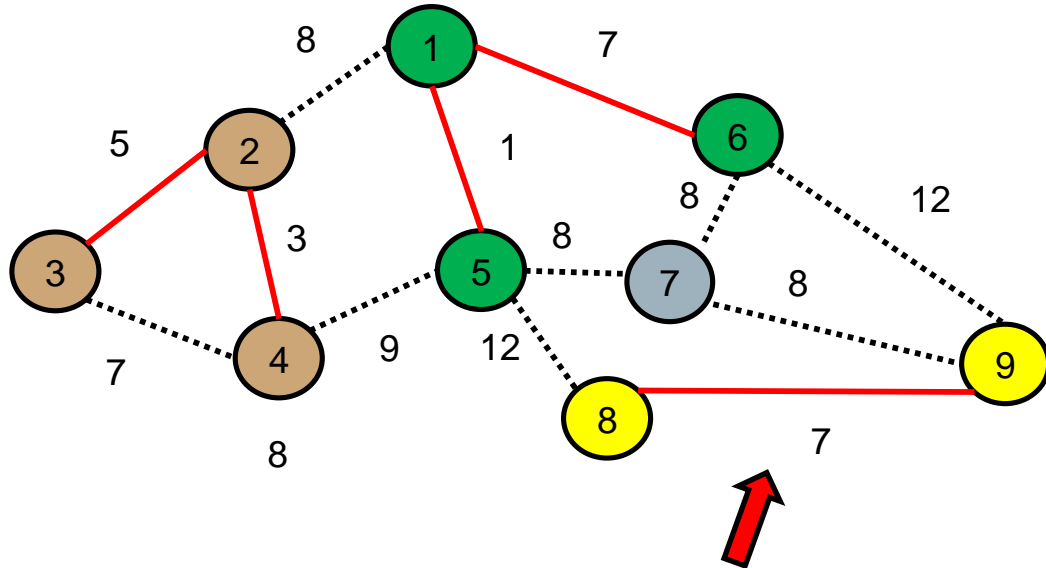


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

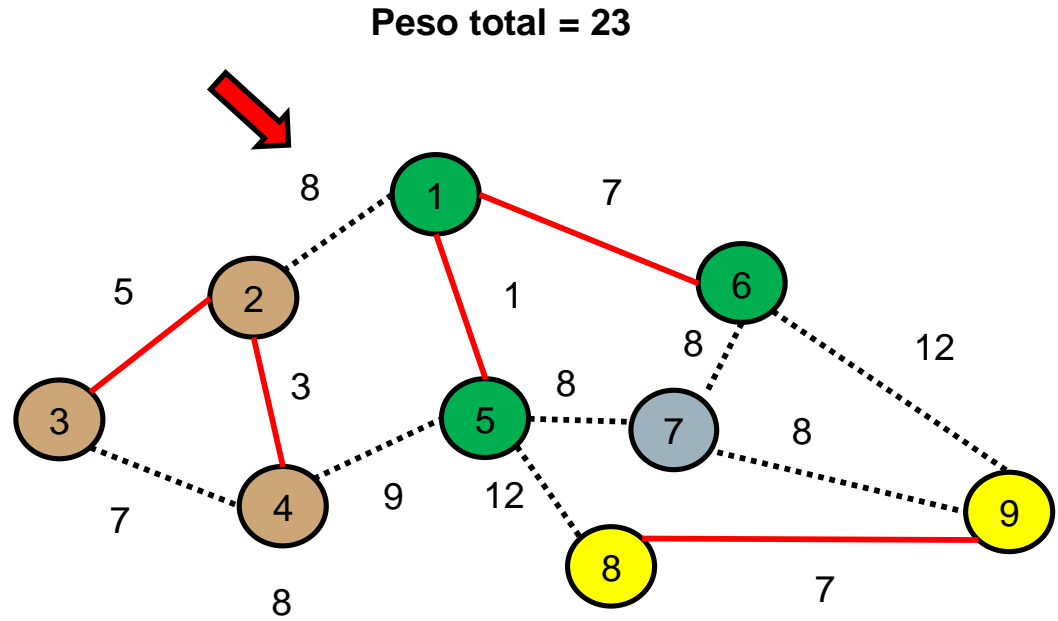
Peso total = 23



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



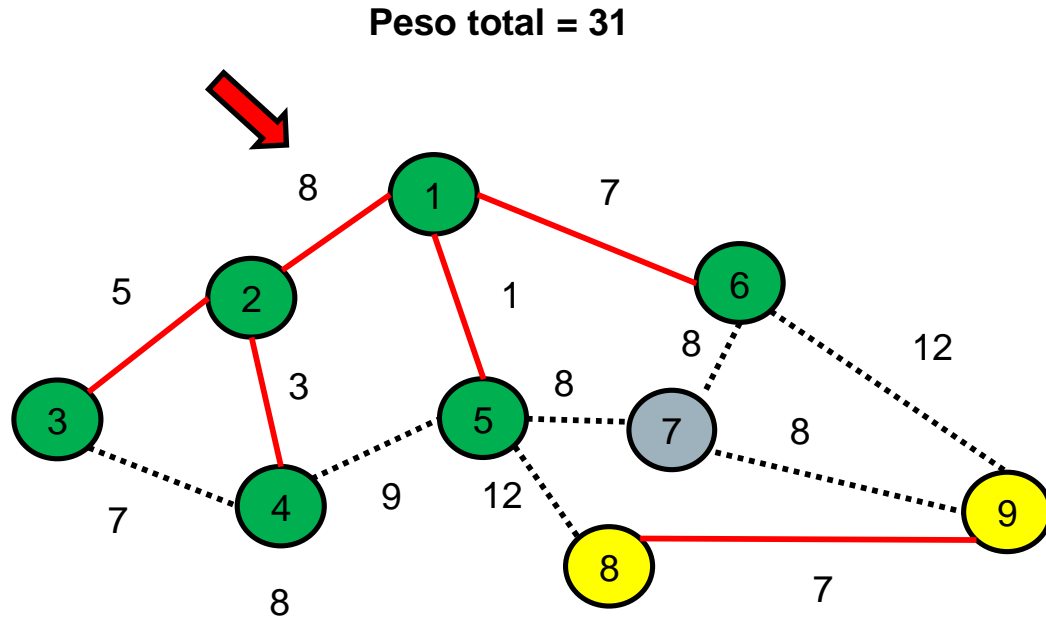
Ejemplo – Algoritmo de Kruskal



| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

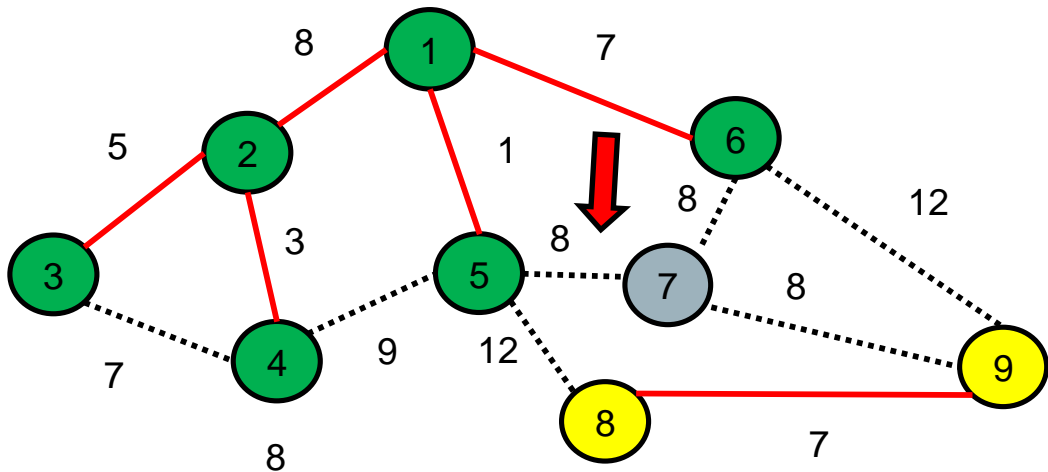


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 31

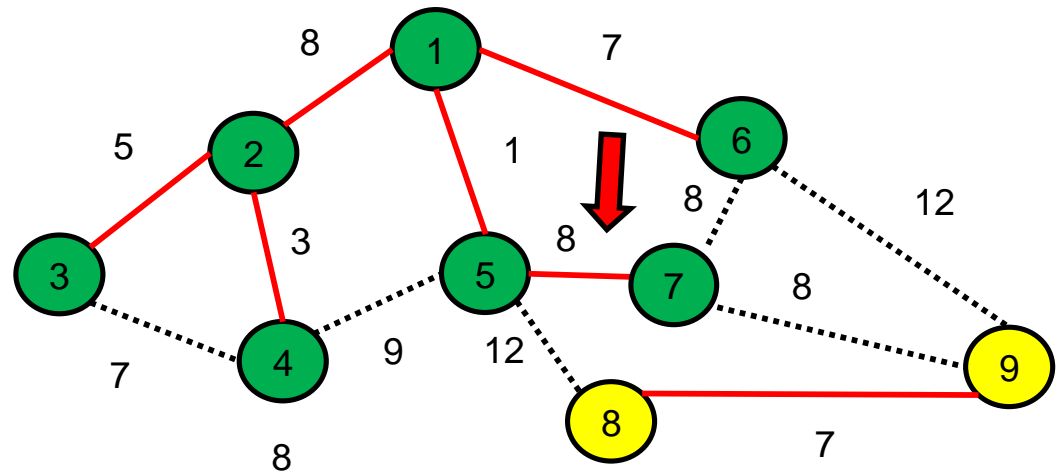


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 39

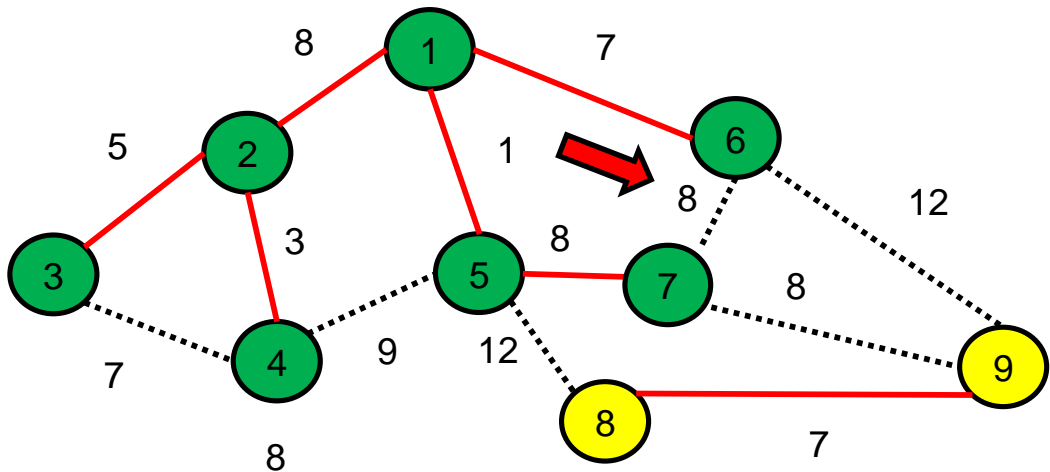


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 39

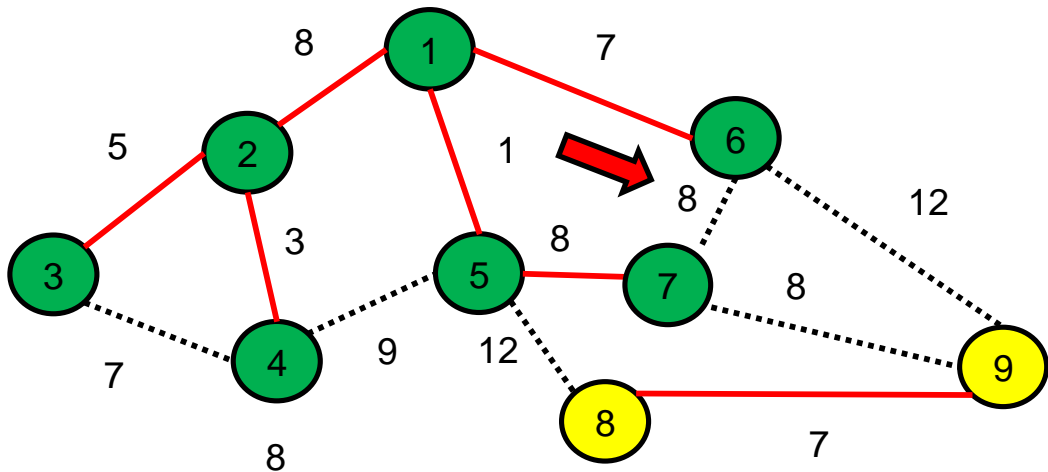


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 39

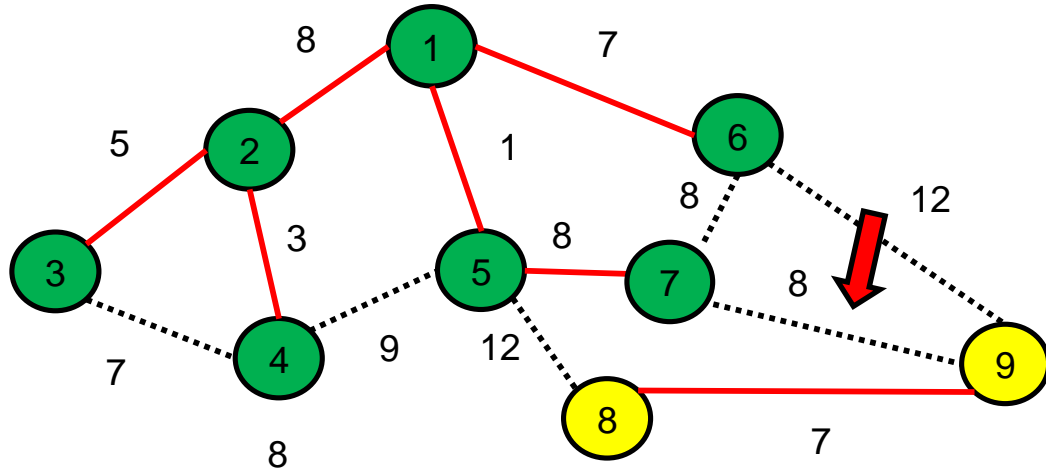


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 39

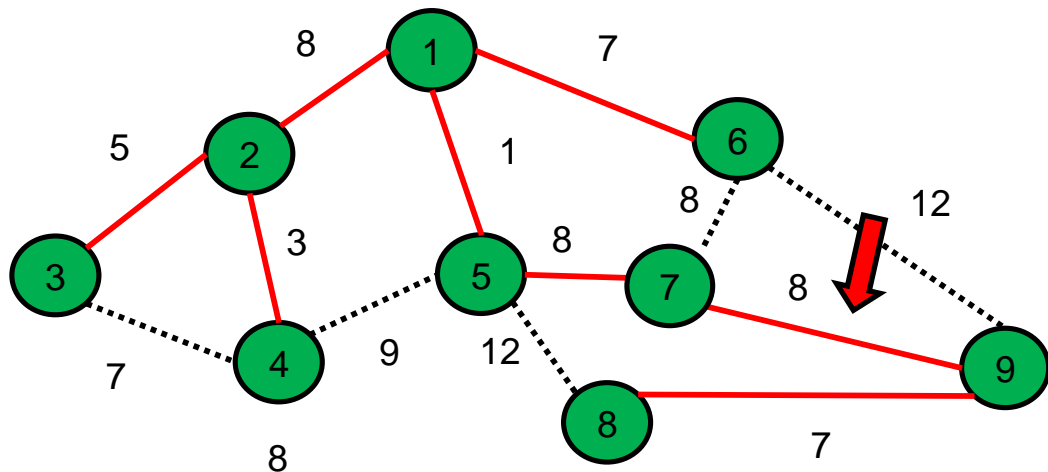


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 47

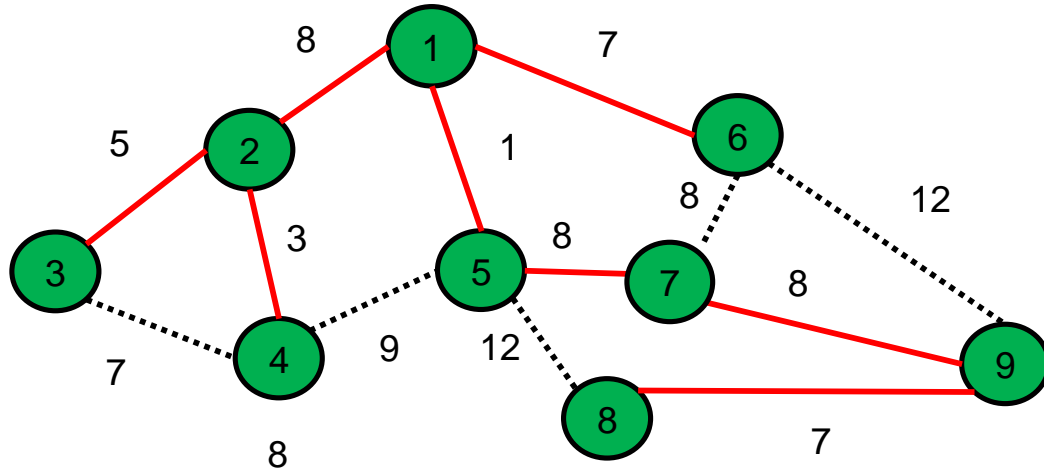


| Arista | Peso |
|---------|------|
| (1 , 5) | 1 |
| (2, 4) | 3 |
| (2, 3) | 5 |
| (1, 6) | 7 |
| (3, 4) | 7 |
| (8 , 9) | 7 |
| (1, 2) | 8 |
| (5, 7) | 8 |
| (6, 7) | 8 |
| (7, 9) | 8 |
| (4, 5) | 9 |
| (5, 8) | 12 |
| (6, 9) | 12 |



Ejemplo – Algoritmo de Kruskal

Peso total = 47

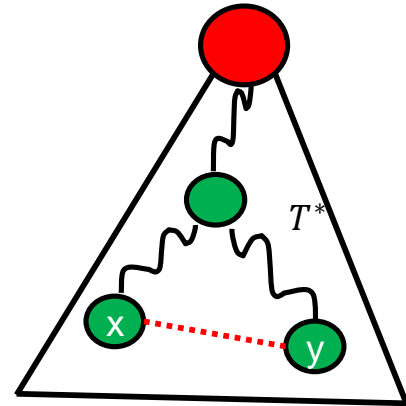
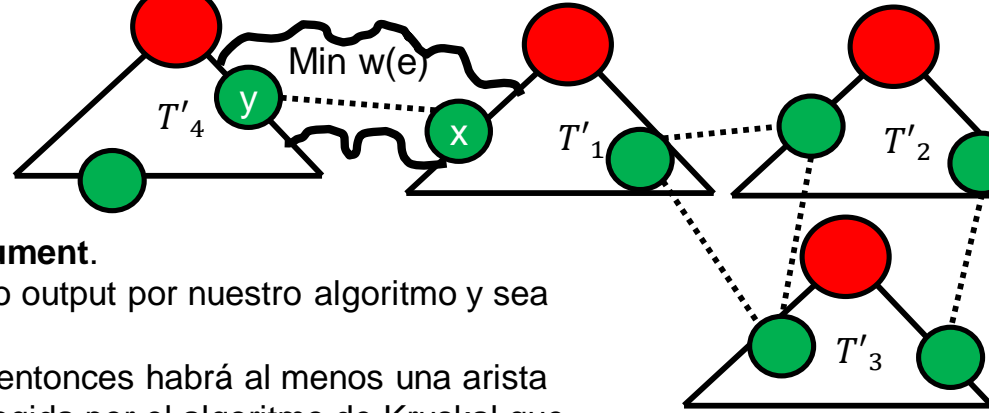


| Arista | Peso | |
|---------|------|---|
| (1 , 5) | 1 | ✓ |
| (2, 4) | 3 | ✓ |
| (2, 3) | 5 | ✓ |
| (1, 6) | 7 | ✓ |
| (3, 4) | 7 | ✗ |
| (8 , 9) | 7 | ✓ |
| (1, 2) | 8 | ✓ |
| (5, 7) | 8 | ✓ |
| (6, 7) | 8 | ✗ |
| (7, 9) | 8 | ✓ |
| (4, 5) | 9 | ✗ |
| (5, 8) | 12 | ✗ |
| (6, 9) | 12 | ✗ |

Proof of correctness

Para probar optimalidad, utilizaremos el **Exchange Argument**.

- 1) **Define las soluciones:** Sea T el árbol dado como output por nuestro algoritmo y sea T^* un MST
- 2) **Compara soluciones:** Supongamos que $T \neq T^*$, entonces habrá al menos una arista en donde varíen. Sea $e(x, y)$ la primera arista escogida por el algoritmo de Kruskal que no esté en T^* , y sea T' el bosque creado por Kruskal justo antes de agregar e .
- 3) **Intercambia elementos:** Si nosotros agregamos e a T^* produciremos un ciclo. En ese ciclo, debe haber alguna arista $e'(u, v)$ tal que u, v estén en distintos árboles en el bosque T' . Por lo tanto, e' también era candidata a ser elegida por Kruskal cuando se eligió e , así que se debe cumplir que $w(e) \leq w(e')$. Además, $T^* + e - e'$ seguirá siendo un árbol de expansión. De lo anterior se deduce que $w(T^* + e - e') \leq w(T^*)$. Pero como T^* era MST, entonces $T^* + e - e'$ también es MST
- 4) **Itera:** Cada vez que hacemos el procedimiento anterior producimos un MST que tiene una arista más en común con T . Si lo repetimos varias veces, conseguiremos un MST igual a T .



Implementación

El ordenamiento de las aristas es bastante simple utilizando la función merge sort de C++ y se hace en $O(m \log m)$.

La parte más complicada es saber si, en determinada iteración del algoritmo, un par de nodos están en una misma componente (mismo árbol).

Sin embargo, la estructura de datos DSU se ajusta perfectamente a este requerimiento. La complejidad del DSU es $O(\alpha(n))$ por operación. Como se hacen a lo más m consultas, la complejidad de esta parte sería $O(m \alpha(n))$

Complejidad total : $O(m \log m + m \alpha(n)) = O(m \log m) = O(m \log n)$. La misma de Prim.

Implementación

```
struct Edge{
    Long u, v, w;
    Edge() {}
    Edge(Long u, Long v, Long w) : u(u) , v(v) , w(w) {}
    bool operator <(Edge const &other) const{
        return w < other.w;
    }
};
```





```
struct Graph{
    vector<Edge> edges;
    vector<pair<Long, Long>> tree[MX]; //node, weight → MST

    void clear(Long n) { //O(n)
        for (int i = 0; i < n; i++) {
            tree[i].clear();
        }
        edges.clear();
    }

    void addEdge(Long u, Long v, Long w) {
        edges.push_back(Edge(u , v , w));
    }

    Long getMST(Long n) { //O(mlogn)
        Long totalWeight = 0;
        dsu.build(n);
        sort(edges.begin(), edges.end());
        for (Edge e : edges) {
            if (dsu.find(e.u) != dsu.find(e.v)) {
                totalWeight += e.w;
                tree[e.u].push_back({e.v, e.w});
                tree[e.v].push_back({e.u, e.w});
                dsu.join(e.u, e.v);
            }
        }
        return totalWeight;
    }
} G;
```

Contenido

| | |
|-------------------------|---|
| 1. Conceptos básicos |  |
| 2. Algoritmo de Prim |  |
| 3. Algoritmo de Kruskal |  |
| 4. Propiedades |  |

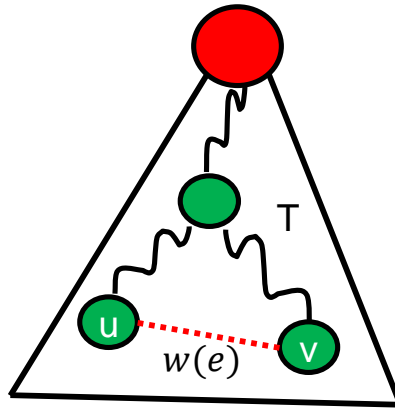
Propiedades

- Si los pesos de las aristas son todos distintos, el MST es único
- Los algoritmos de Kruskal y Prim funcionan para **pesos reales**, incluso con negativos y con pesos no enteros.
- Si todos los pesos son positivos, se cumple que el MST también **minimiza el producto de los pesos de las aristas**. Justificación: Si reemplazas cada peso por su logaritmo, el ordenamiento de estas aristas no cambiará ya que si $0 < x \leq y \Rightarrow \log x \leq \log y$. Como el orden es el mismo, Kruskal elegirá las mismas aristas y se formará el mismo árbol. Además el MST de los logaritmos minimizará la sumatoria de logaritmos $\sum \log w_i = \log \prod w_i$, lo cual minimiza el producto de pesos del grafo original. (Ojo: No es necesario que en la implementación reemplaces por logaritmo, es solo para la demostración).
- El **Maximum Spanning Tree** se puede obtener reemplazando cada arista con su negativo y dando como output el negativo de la suma de pesos. Otra forma es aplicando, por ejemplo, Kruskal pero haciendo el ordenamiento de mayor a menor (no creciente).

Propiedades

- Sea un undirected weighted graph G con MST T . Si una arista $e(u, v)$ de G no está en T , entonces $w(e) \geq w(e')$ para cualquier arista e' que esté en el path entre u, v .

Justificación: Por reducción al absurdo, si $w(e) < w(e')$, entonces $T' = T + e - e'$ también es un árbol de expansión y cumple que $w(T') < w(T)$. Pero entonces T no sería MST y llegamos a una contradicción.

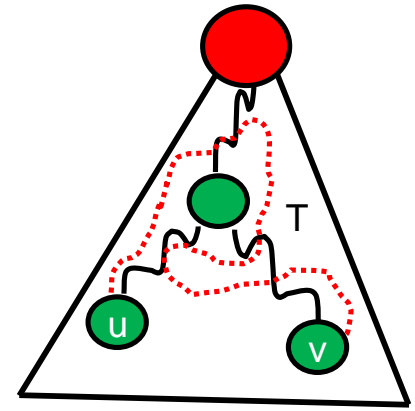


Propiedades

- Sea un undirected weighted graph G con MST T . Para cualquier par de vértices (u, v) el único path entre u, v en el MST es el camino que cuya **máxima arista es mínima** en comparación con cualquier otro camino entre u, v en el grafo G .

Justificación 1: Como el algoritmo de Kruskal es correcto, y el algoritmo de Kruskal empieza desde la arista de menor peso hacia adelante y trata de añadir la arista siempre que no produzca un ciclo, entonces en el momento en que consigues que u, v estén en la misma componente, lo habrás hecho de forma que la máxima arista se minimice.

Justificación 2: Por reducción al absurdo supongamos que existiera otro camino cuya máxima arista sea menor que la que hay en el camino entre u, v en el MST. Este camino contiene aristas que no están en MST aunque también puede contener algunas que sí estén en el MST. Sin embargo, cada arista que NO pertenece al MST no contribuye a mejorar la respuesta debido a la propiedad anterior. Por lo tanto cada arista $e'(x, y)$ que NO pertenece al MST puede ser reemplazada por el camino directo entre x, y en el MST. Y la unión de todos estos caminos siempre tiene que pasar por el camino directo entre u y v .



Referencias

- ❑ CP-algorithms: https://cp-algorithms.com/graph/mst_prim.html
- ❑ CP-algorithms: https://cp-algorithms.com/graph/mst_kruskal.html
- ❑ CP-algorithms: <https://cp-algorithms.com/graph/kirchhoff-theorem.html>
- ❑ Princeton : <https://www.cs.princeton.edu/~rs/AlgsDS07/14MST.pdf>
- ❑ University of Toronto: <http://www.cs.toronto.edu/~lalla/373s16/notes/MST.pdf>
- ❑ EPFL: <https://archiveweb.epfl.ch/dcg.epfl.ch//wp-content/uploads/2018/10/GT-2-Trees.pdf>