**Algoritmo de spanning tree**

**Alejandro Hernández Gómez**

En [comunicaciones](https://es.wikipedia.org/wiki/Comunicaci%C3%B3n_en_red), STP (del inglés *Spanning Tree Protocol*) es un protocolo de red de nivel 2 del [modelo OSI](https://es.wikipedia.org/wiki/Modelo_OSI) (capa de enlace de datos). Su función es la de gestionar la presencia de bucles en topologías de red debido a la existencia de enlaces redundantes (necesarios en muchos casos para garantizar la disponibilidad de las conexiones). El protocolo permite a los dispositivos de interconexión activar o desactivar automáticamente los enlaces de conexión, de forma que se garantice la eliminación de bucles. STP es transparente a las estaciones de usuario.

Está basado en un algoritmo diseñado por [Radia Perlman](https://es.wikipedia.org/wiki/Radia_Perlman) mientras trabajaba para DEC. Hay 2 versiones del STP: la original (DEC STP) y la estandarizada por el IEEE ([IEEE 802.1D](https://es.wikipedia.org/wiki/IEEE_802.1D)), que no son compatibles entre sí. En la actualidad, se recomienda utilizar la versión estandarizada por el IEEE.

Existen múltiples variantes del STP debido, principalmente, al tiempo que tarda en converger el algoritmo utilizado. Una de estas variantes es el [Rapid Spanning Tree Protocol](https://es.wikipedia.org/wiki/Rapid_Spanning_Tree_Protocol), estándar IEEE 802.1D-2004 que hoy en día ha reemplazado el uso del STP original. 2012 [IEEE 802.1aq](https://es.wikipedia.org/wiki/IEEE_802.1aq) fue aprobado como un estándar para reemplazar [IEEE 802.1D](https://es.wikipedia.org/wiki/IEEE_802.1D), [IEEE 802.1w](https://es.wikipedia.org/wiki/IEEE_802.1w), [IEEE 802.1s](https://es.wikipedia.org/w/index.php?title=IEEE_802.1s&action=edit&redlink=1)

El algoritmo transforma una red física con forma de malla, en la que existen bucles, por una red lógica en forma de árbol (libre de bucles). Los [puentes](https://es.wikipedia.org/wiki/Puente_de_red) se comunican mediante mensajes de configuración llamados [Bridge Protocol Data Units](https://es.wikipedia.org/wiki/Bridge_Protocol_Data_Units) (BPDU).

El protocolo establece *identificadores por puente* y elige el que tiene la prioridad más alta (el número más bajo de prioridad numérica), como el *puente raíz* (Root Bridge). Este puente raíz establecerá el camino de menor coste para todas las redes; cada puerto tiene un parámetro configurable: el [Span path cost](https://es.wikipedia.org/w/index.php?title=Span_path_cost&action=edit&redlink=1" \o "Span path cost (aún no redactado)). Después, entre todos los puentes que conectan un segmento de red, se elige un *puente designado*, el de menor coste (en el caso que haya el mismo coste en dos puentes, se eige el que tenga el menor identificador "dirección MAC"), para transmitir las tramas hacia la raíz. En este puente designado, el puerto que conecta con el segmento, es el *puerto designado* y el que ofrece un camino de menor coste hacia la raíz, el *puerto raíz*. Todos los demás puertos y caminos son bloqueados, esto es en un estado ya estacionario de funcionamiento.

Los estados en los que puede estar un puerto son los siguientes:

* Bloqueo: En este estado se pueden recibir BPDU's pero no las enviará. Las tramas de datos se descartan y no se actualizan las tablas de direcciones MAC (mac-address-table). Los switch comienzan en este estado ya que si realizan envíos (forwarding) podrían estar generando un loop o bucle.
* Escucha: A este estado se llega desde *Bloqueo*. En este estado, los switches determinan si existe alguna otra ruta hacia el puente raíz. En el caso que la nueva ruta tenga un coste mayor, se vuelve al estado de Bloqueo. Las tramas de datos se descartan y no se actualiza la tabla de direcciones MAC (mac-address-table). Se procesan las BPDU.
* Aprendizaje: A este estado se llega desde *Escucha*. Las tramas de datos se descartan pero ya se actualizan las tablas de direcciones MAC (aquí es donde se aprenden por primera vez). Se procesan las BPDU.
* Envío: A este estado se llega desde *Aprendizaje*, en este estado el puerto puede enviar y recibir datos. Las tramas de datos se envían y se actualizan las tablas de direcciones MAC (mac-address-table). Se procesan las BPDU.
* Desactivado: A este estado se llega desde cualquier otro. Se produce cuando un administrador deshabilita el puerto o éste falla. No se procesan las BPDU.

Para complementar esta pequeña investigación mostraré una integración del *Kruskal*

public class KruskalMST {

private static final double FLOATING\_POINT\_EPSILON = 1E-12;

private double weight; *// weight of MST*

private Queue<Edge> mst = new Queue<Edge>(); *// edges in MST*

*/\*\**

*\* Compute a minimum spanning tree (or forest) of an edge-weighted graph.*

*\** @param *G the edge-weighted graph*

*\*/*

public **KruskalMST**(EdgeWeightedGraph G) {

*// more efficient to build heap by passing array of edges*

MinPQ<Edge> pq = new MinPQ<Edge>();

for (Edge e : G.**edges**()) {

pq.**insert**(e);

}

*// run greedy algorithm*

UF uf = new **UF**(G.**V**());

while (!pq.**isEmpty**() && mst.**size**() < G.**V**() - 1) {

Edge e = pq.**delMin**();

int v = e.**either**();

int w = e.**other**(v);

if (!uf.**connected**(v, w)) { *// v-w does not create a cycle*

uf.**union**(v, w); *// merge v and w components*

mst.**enqueue**(e); *// add edge e to mst*

weight += e.**weight**();

}

}

*// check optimality conditions*

assert **check**(G);

}

*/\*\**

*\* Returns the edges in a minimum spanning tree (or forest).*

*\** @return *the edges in a minimum spanning tree (or forest) as*

*\* an iterable of edges*

*\*/*

public Iterable<Edge> **edges**() {

return mst;

}

*/\*\**

*\* Returns the sum of the edge weights in a minimum spanning tree (or forest).*

*\** @return *the sum of the edge weights in a minimum spanning tree (or forest)*

*\*/*

public double **weight**() {

return weight;

}

*// check optimality conditions (takes time proportional to E V lg\* V)*

private boolean **check**(EdgeWeightedGraph G) {

*// check total weight*

double total = 0.0;

for (Edge e : **edges**()) {

total += e.**weight**();

}

if (Math.**abs**(total - **weight**()) > FLOATING\_POINT\_EPSILON) {

System.err.**printf**("Weight of edges does not equal weight(): %f vs. %f\n", total, **weight**());

return false;

}

*// check that it is acyclic*

UF uf = new **UF**(G.**V**());

for (Edge e : **edges**()) {

int v = e.**either**(), w = e.**other**(v);

if (uf.**connected**(v, w)) {

System.err.**println**("Not a forest");

return false;

}

uf.**union**(v, w);

}

*// check that it is a spanning forest*

for (Edge e : G.**edges**()) {

int v = e.**either**(), w = e.**other**(v);

if (!uf.**connected**(v, w)) {

System.err.**println**("Not a spanning forest");

return false;

}

}

*// check that it is a minimal spanning forest (cut optimality conditions)*

for (Edge e : **edges**()) {

*// all edges in MST except e*

uf = new **UF**(G.**V**());

for (Edge f : mst) {

int x = f.**either**(), y = f.**other**(x);

if (f != e) uf.**union**(x, y);

}

*// check that e is min weight edge in crossing cut*

for (Edge f : G.**edges**()) {

int x = f.**either**(), y = f.**other**(x);

if (!uf.**connected**(x, y)) {

if (f.**weight**() < e.**weight**()) {

System.err.**println**("Edge " + f + " violates cut optimality conditions");

return false;

}

}

}

}

return true;

}

*/\*\**

*\* Unit tests the {*@code *KruskalMST} data type.*

*\**

*\** @param *args the command-line arguments*

*\*/*

public static void **main**(String[] args) {

In in = new **In**(args[0]);

EdgeWeightedGraph G = new **EdgeWeightedGraph**(in);

KruskalMST mst = new **KruskalMST**(G);

for (Edge e : mst.**edges**()) {

StdOut.**println**(e);

}

StdOut.**printf**("%.5f\n", mst.**weight**());

}

}