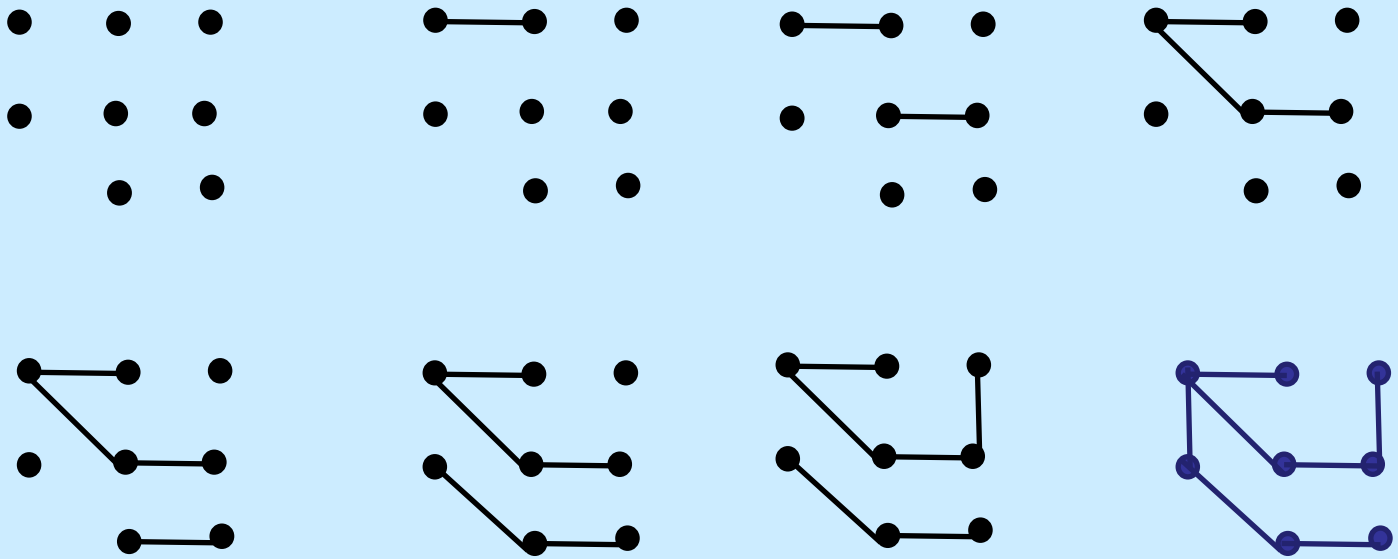
Tema 7. Árboles. Árboles Generadores.

1. **La propiedad 'ser árbol'**
2. **Árboles generadores de un grafo conexo**
3. **El problema de árbol generador de mínimo coste**

Árboles

Los árboles son un tipo de grafo no dirigido con una estructura ya de por sí **óptima**. Veamos por qué…

Primer problema a resolver: partiendo de un grafo vacío de orden n, añadir aristas con la condición de evitar ciclos…

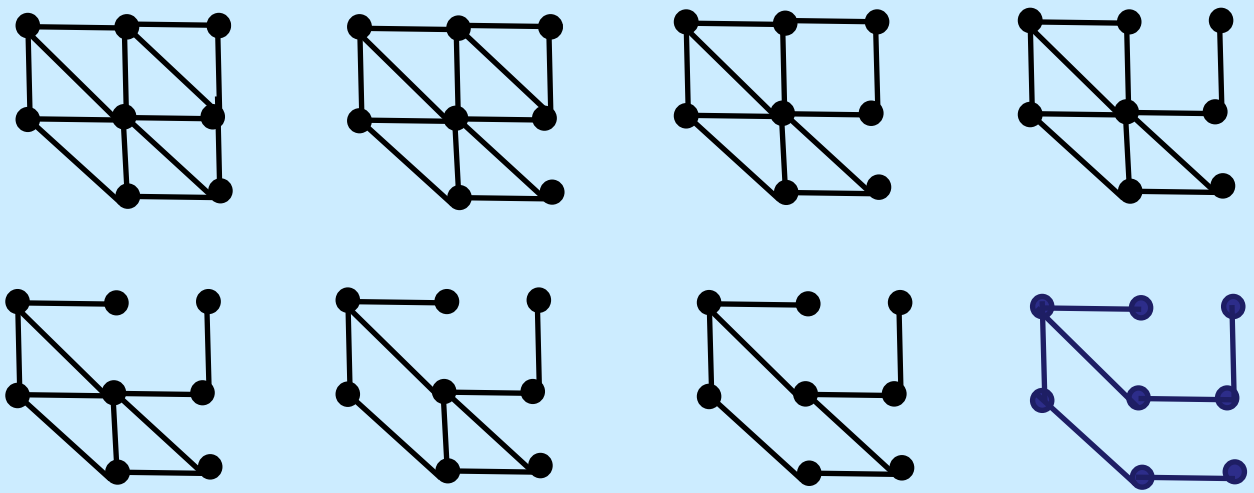


Ya no es posible continuar añadiendo aristas sin romper la propiedad **ser acíclico**…

Árboles

Los árboles son un tipo de grafo no dirigido con una estructura ya de por sí **óptima**. Veamos por qué…

Segundo problema a resolver: partiendo de un grafo conexo de orden n, eliminar aristas con la condición mantener la conexidad…

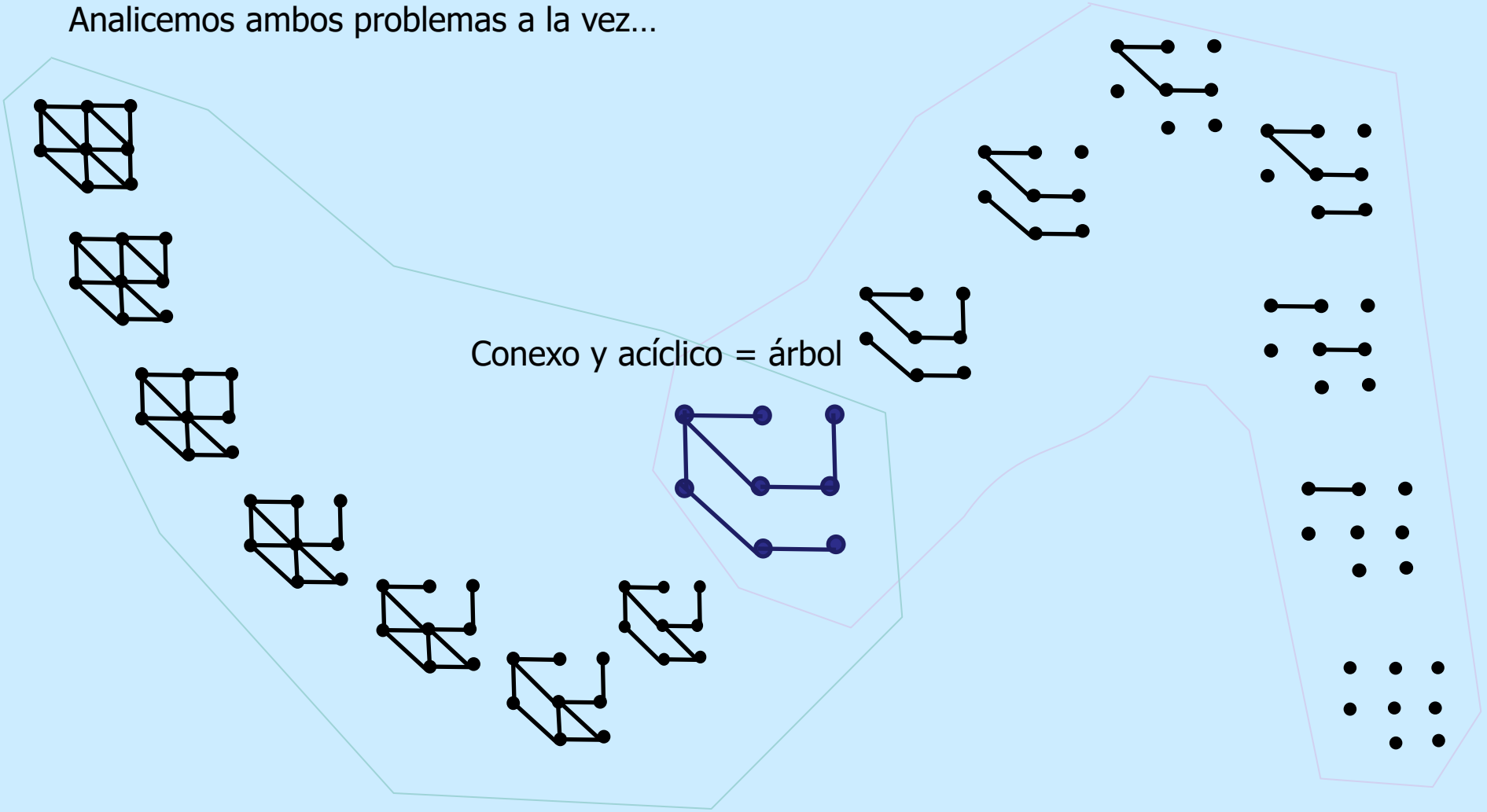


Ya no es posible continuar eliminando aristas sin romper la conexidad del grafo…

Árboles

Secuencia de grafos acíclicos

Analicemos ambos problemas a la vez...



Secuencia de grafos conexos

Árboles

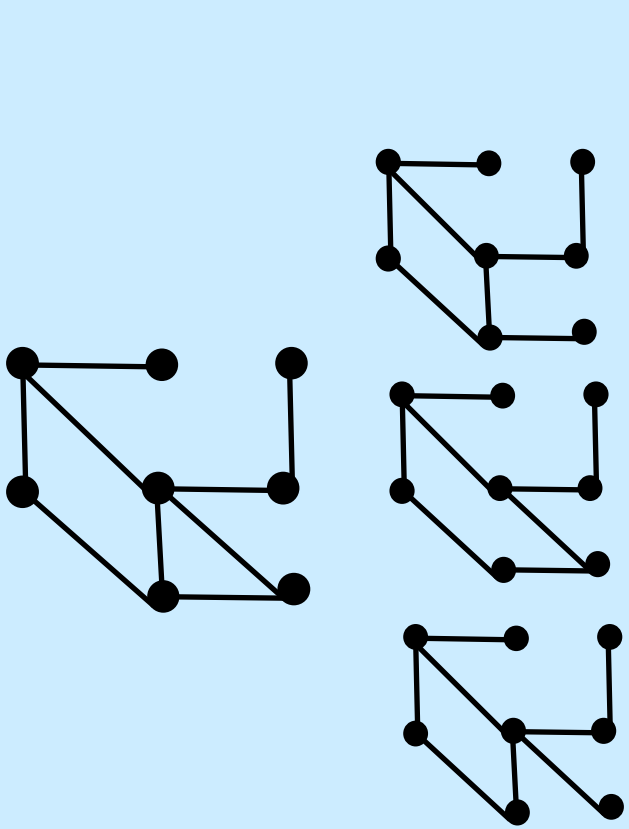
Por tanto, el **árbol** es un grafo en **equilibrio**: suficientes aristas para conectar todos los nodos (conexos) pero no demasiadas como para ser redundante (acíclico). Es una estructura de conexión óptima

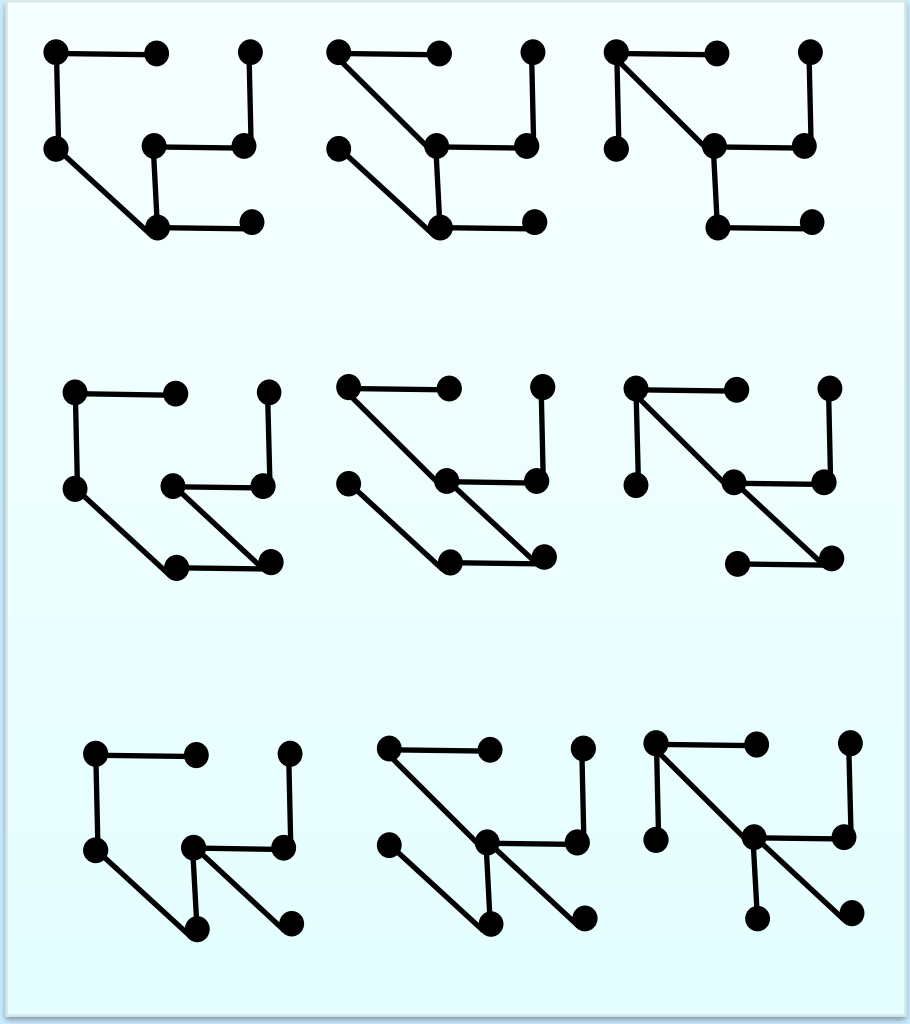
Definiciones equivalentes: un árbol es un grafo

1. Conexo y acíclico
2. Acíclico maximal (si incorporamos una arista más deja de ser acíclico)
3. Conexo minimal (si quitamos una arista deja de ser conexo)
4. Conexo con n-1 aristas
5. Acíclico con n-1 aristas
6. Para cada par de nodos existe una **única** cadena que los une.

Árboles generadores

Problema: dado un grafo conexo, **¿qué subgrafos contiene que sean árboles?**





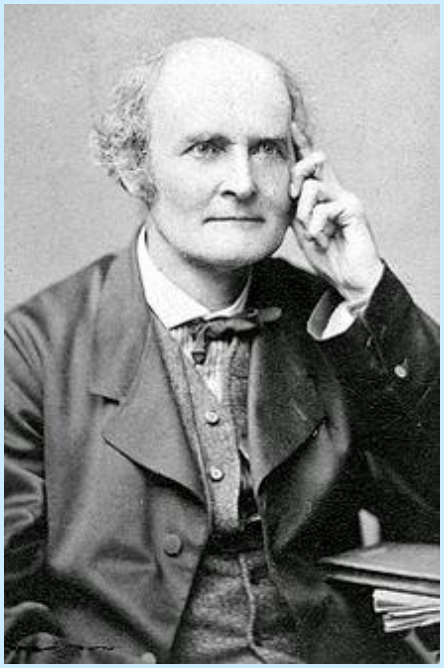
Árboles generadores

Dado un grafo conexo, G=(V,A), decimos que un subconjunto de aristas es un **árbol generador** de G si es un árbol de n nodos.

Cuestiones a tener en cuenta:

- ¿Porqué el grafo de partida ha de ser conexo?
- Dado un grafo completo de orden n, ¿cuántos árboles generadores pueden obtenerse?

$$n^{n-2}$$



Árboles generadores

Dado un grafo conexo, G=(V,A), decimos que un subconjunto de aristas es un **árbol generador** de G si es un árbol de n nodos.

Cuestiones a tener en cuenta:

- Si tardásemos una millonésima de segundo es generar cada árbol generador mediante un algoritmo, ¿cuánto tardaríamos en construir todos los árboles generadores de un grafo completo de 5 nodos? ¿Y de 20 nodos?

	Árboles en un segundo 1.000.000						Edad de la tierra en miles de millones de años 4.467
n	árboles generadores	segundos	horas	días	años	miles de años	miles de millones de años
4	16	**0,000016**					
5	125	**0,000125**					
6	1296	**0,001296**					
10	100.000.000	**100**					
15	1.946.195.068.359.370	1.946.195.068,4	540.609,7	22.525,4	**61,7**		
20	2,62144E+23	2,62144E+17	7,28E+13	3,03E+12	8,31E+09	8,31E+06	**8,3**
50	3,55271E+81	3,55271E+75	9,86865E+71	4,11194E+70	1,12579E+68	1,12579E+65	1,13E+59
100	1E+196	1E+190	2,7778E+186	1,1574E+185	3,1688E+182	3,1688E+179	3,17E+173

Árboles generadores

¿Cómo construir árboles generadores de un grafo conexo?
Sea G=(V,A) un grafo no dirigido conexo

Método K
T=∅
Mientras en T no haya n-1 aristas hacer
    Sea e una arista tal que T∪{e} es acíclico
    T= T∪{e}

Método P
T=∅
M={1}
Mientras en T no haya n-1 aristas hacer
    Sea e una arista con un extremo en M, y el otro, j, en V-M
    T= T∪{e}
    M=M ∪{j}

Ambos métodos construyen un árbol generador arbitrario… ¿por qué estamos seguros?

Árboles generadores

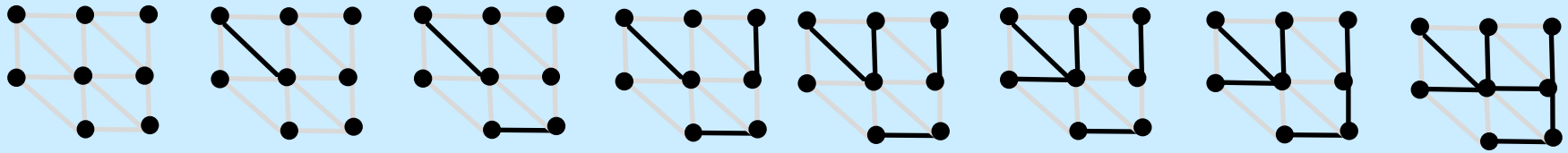
Método K

T=∅

Mientras en T no haya n-1 aristas hacer

Sea e una arista tal que T∪{e} es acíclico

T= T∪{e}



¿Es finalmente T un árbol generador de G?

Método K

T=∅

Mientras en T no haya n-1 aristas hacer

Sea e una arista **cuyos extremos están en componentes conexas distintas**

T= T∪{e}

Árboles generadores

Método P

T=∅

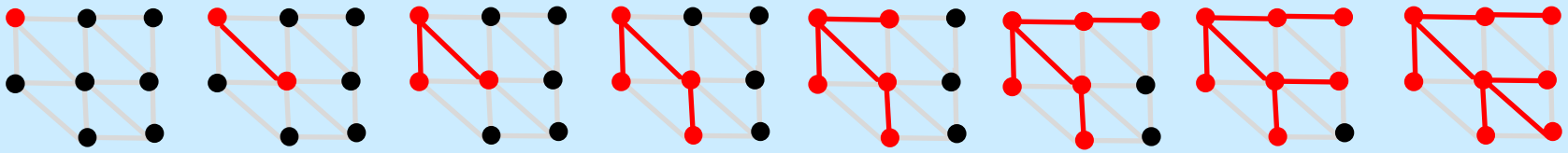
M={1}

Mientras en T no haya n-1 aristas hacer

Sea e una arista con un extremo en M, y el otro, j, en V-M

T= T∪{e}

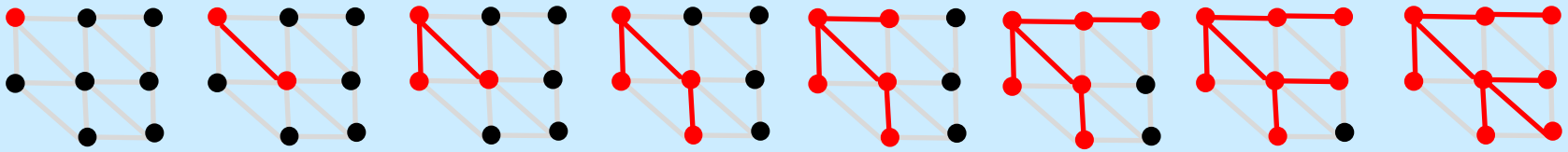
M=M ∪{j}



¿Es finalmente T un árbol generador de G?

Árboles generadores de mínimo coste

Para cada uno de los métodos anteriores, la identificación de la arista a introducir en T debía verificar condiciones 'estructurales' que nos aseguraran que finalmente T fuera un árbol generador…
En cada proceso de selección, había **más de una** arista candidata…
¿Cómo elegir 'la **mejor**' entre ellas?



Sea G=(V,A) un grafo conexo 'pesado', esto es, **cada arista tiene asignado un peso o coste**, que se asocia al **gasto que aporta o implica considerarlo dentro del árbol generador que estoy construyendo**…
Para cada $e \in A$, su coste de denota por $\omega(e)$ y es, generalmente, un número de $\mathbb{R}$. Linealmente,

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

Si tenemos métodos para construir **todos** los árboles generadores de G, ¿Cómo reescribirlos para que nos construyan **el de menor coste**?

Árboles generadores de mínimo coste

Método K

T=∅

Mientras en T no haya n-1 aristas hacer

Sea e una arista **cuyos extremos están en componentes conexas distintas**

T= T∪{e}

Algoritmo de Kruskal, 1956

T=∅

Mientras en T no haya n-1 aristas hacer

Sea e **la arista de menor coste** cuyos extremos están en componentes conexas distintas

T= T∪{e}

Árboles generadores de mínimo coste

Método P

T=∅
M={1}
Mientras en T no haya n-1 aristas hacer
    Sea e una arista con un extremo en M, y el otro, j, en V-M
    T= T∪{e}
    M=M ∪{j}

Algoritmo de Prim, 1957

T=∅
M={1}
Mientras en T no haya n-1 aristas hacer
    Sea e la arista de menor coste con un extremo en M, y el otro, j, en V-M
    T= T∪{e}
    M=M ∪{j}

Implementando el algoritmo de Kruskal

Algoritmo de Kruskal, 1956

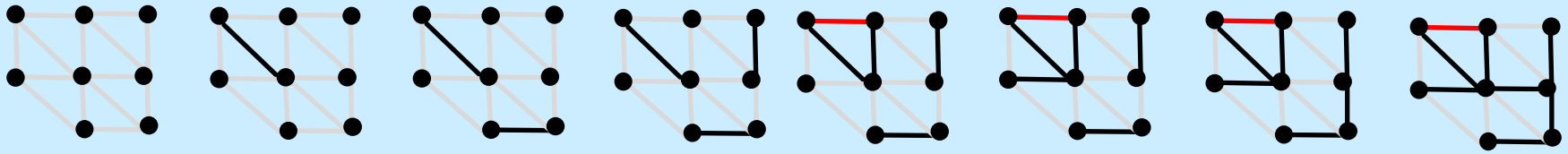
T=∅

Mientras en T no haya n-1 aristas hacer

Sea e la arista de menor coste cuyos extremos están en componentes conexas distintas

T= T∪{e}

¿Cómo simplificar y optimizar lo que hemos de hacer? Pensemos…



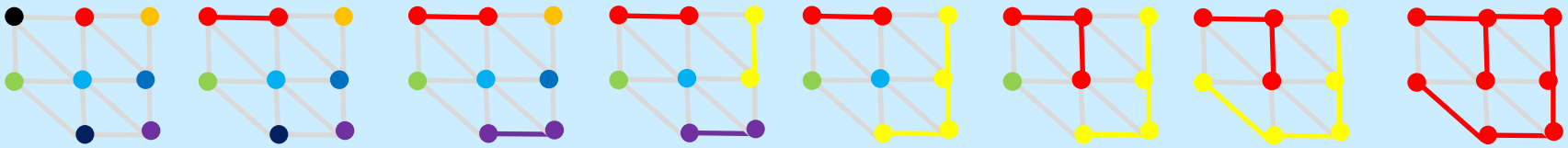
Si una arista forma ciclo y es rechazada,
nunca entrará en la solución más adelante

Simplificamos si ordenamos todas las aristas según su coste o peso, y comprobamos si podemos introducirla en la solución que estamos construyendo

Implementando el algoritmo de Kruskal

Algoritmo de Kruskal, 1956
Ordenar las aristas en orden no decreciente según sus costes
T=∅
Mientras en T no haya n-1 aristas hacer
Sea e la **siguiente** arista cuyos extremos están en componentes conexas distintas
T= T∪{e}

¿Cómo saber si a qué componente conexa pertenece un nodo? Pensemos…

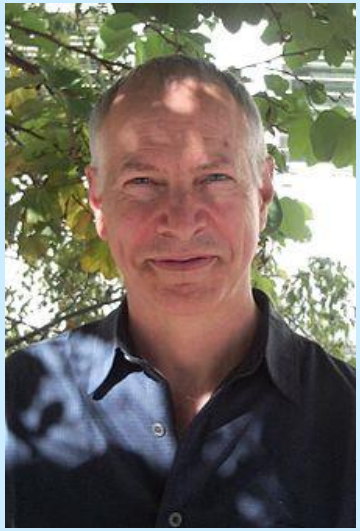


Etiquetando los nodos: si tienen la misma etiqueta, están en la misma componente conexa, si tienen distinta etiqueta, distintas componente…

Implementando el algoritmo de Kruskal

Algoritmo de Kruskal, 1956
Ordenar las aristas en orden no decreciente según sus costes
T=∅
Para todo nodo i de V hacer raiz[i] = i
Mientras en T no haya n-1 aristas hacer
    Sea e=(i,j) la **siguiente** arista
    Si cuyos raiz[i] <> raiz[j] entonces
        T= T∪{e}
        kill = raiz[i]
        Para todo nodo k de V hacer
            Si raiz[k]=kill entonces raiz[k]=raiz[j]

+info sobre cómo gestionar particiones de V: busca Tarjan



Implementando el algoritmo de Prim

Algoritmo de Prim, 1957
T=∅
M={1}
Mientras en T no haya n-1 aristas hacer
    Sea e la arista de menor coste con un extremo en M, y el otro, j, en V-M
    T= T∪{e}
    M=M ∪{j}

Un completo cambio de estrategia: no metemos aristas de bajo coste, sino nodos de bajo coste, ¿cómo?

Mantenemos la estructura llamada etiqueta distancia, que nos permite saber, en cada momento, cuanto nos cuesta meter ese nodo en M…
Para ello, guardamos en el vector coste ese valor, inicializándolo a infinito en el caso de todos los nodos, menos el nodo 1, que es el que metemos en M por defecto…
A la vez, debemos mantener la estructura pred, que guarda el nodo de M que metería el nodo a estudio en M de la forma más económica…
Con estos dos vectores se implementa el algoritmo de Prim…

Implementando el algoritmo de Prim

Algoritmo de Prim, 1957

```
T=∅
Para todo nodo i de V hacer coste[i] = infinito
M={1}
coste[1]=0
pred[1]=1
Mientras en T no haya n-1 aristas hacer
        sea u el último nodo que entró en M
        para todo j adyacente a u hacer
                si coste[j] es peor que w(u, j) entonces
                        coste[j]=w(u,j)
                        pred[j] = u
        sea u = arg min coste [j] para todo j en V-M
        M = M ∪ {u}
        T = T ∪ {(u, pred[u])}
```