TEMA 4: ALGORITMOS DE GRAFOS.

4.0.- Definicións.
4.1.- Representación de grafos.
4.2.- Ordenación topolóxica.
4.3.- Árbore de expansión mínima (árbore expandida).
4.3.1.- Algoritmo de Kruskal.
4.3.2.- Algoritmo de Prim.
4.4. Algoritmo do camiño mais curto (Dijkstra).

4.0.- Definicións.

Definición de grafo: G = (V,A) onde V é un conxunto de *vértices* e A é un conxunto de *aristas*.

Cada arista é un par (v,w) onde v, w ∈ V.

Se os pares que forman A son ordenados -i.e. (v,w)≠(w,v)-, o grafo é *dirixido*; se polo contrário non son ordenados -i.e. (v,w)=(w,v)- entón o grafo é *non dirixido*.

O vértice w é *adxacente* a v ⇔ (v,w) ∈ A (grafo dirixido). Nos grafos non dirixidos é indistinto (v,w) ou (w,v).

As aristas poden ser tamén ternas formadas polo vértice orixen, destino e peso da arista que leva do primeiro ao segundo (*grafo pesado ou ponderado*).

Un *camiño* nun grafo é unha secuência de vértices w1, w2, ..., wn tal que cada par consecutivo de vértices está unido por unha arista existente, ∀ $(w_i,w_{i+1}) \in A$ con $1 \leq i < n$

A *lonxitude do camiño* é o número de aristas do camiño (n-1 na notación anterior).

Permiten-se camiños de un vértice a si mesmo; se este camiño non ten aristas, entón o camiño ten lonxitude cero.

Se o grafo contén unha arista (v,v) de un vértice a si mesmo, entón o camiño v, v coñece-se como *ciclo*.

Un *camiño simples* é un camiño tal que *todos os vértices son distintos* agás o primeiro e o derradeiro que poden ser o mesmo.

En *grafos dirixidos*, un *ciclo* é un *camiño de lonxitude mínima 1* tal que *o primeiro vértice e o derradeiro son o mesmo* $w_1=w_n$; e este ciclo é simples (non ten vértices intermédios repetidos) se o camiño é simples. Para grafos non dirixidos para que un camiño sexa considerado ciclo **ademais** requerimos que as aristas sexan diferentes.

Exemplo:

grafo dirixido: u,v,u é ciclo porque (u,v) e (v,u) son aristas distintas. Un camiño nun grafo dirixido para ser ciclo debe cumprir: lonxitude mínima 1, e primeiro vértice = derradeiro vértice.

grafo non dirixido: u,v,u non é ciclo porque (u,v) e (v,u) son a mesma arista. Un camiño nun grafo non dirixido debe cumprir: lonxitude mínima 1, e primeiro vértice = derradeiro vértice e **ademais** as aristas deben ser todas distintas. As duas primeiras condicións cumpren-se pero a terceira non.

Un grafo dirixido acíclico (GDA) é un grafo dirixido sen ciclos.

Un grafo non dirixido é *conexo* se hai un camiño desde calquer vértice a calquer outro.

Un grafo dirixido é *fortemente conexo* se hai un camiño desde calquer vértice a calquer outro.

Un grafo dirixido é *dèbilmente conexo* se o grafo subxacente (sen dirección nas aristas) é conexo.

Un grafo *completo* é un grafo no que hai *unha arista* entre calquer par de vértices.

Exemplo de aplicación de grafos: modelizar o plano do metro. Verificar se o grafo é fortemente conexo: se un viaxeiro pode ir desde calquer estación a calquer outra. Determinar o traxecto óptimo.

4.1.- Representación de grafos.

Imos considerar grafos dirixidos (os non dirixidos representan-se de un xeito semellante). Supomos que os nodos están numerados cos naturais do 1 en adiante.

MATRIZ DE ADXACÊNCIA

Unha representación sinxela é a matriz de adxacência, unha matriz bidimensional cuadrada de orden $|V|x|V|$ onde a[u,v]=1 se hai arista de u a v, $(u,v) \in A$; e a[u,v]=0 se non hai arista de u a v, $(u,v) \notin A$. Se o grafo é pesado cada posición da matriz de adxacência ten un valor de entre os seguintes: $+\infty$ ou $-\infty$ ou 0 para representar que non hai arista; calquer valor distinto do valor "nulo" escollido, indicando o peso da arista se esta existe. O ∞ indica un valor extremo que non pode ser baixo nengunha circunstância un peso válido para nengunha arista e que polo tanto queda claramente especificado como valor nulo.

O problema que se plantexa é o requerimento de espazo: note-se que para esta representación é preciso un espazo $\Theta(|V|^2)$, o cal é excesivo se o grafo é disperso (disperso é o contrário de denso).

Asi que usamos outra representación para o caso de que o grafo sexa disperso:

LISTAS DE ADXACÊNCIA

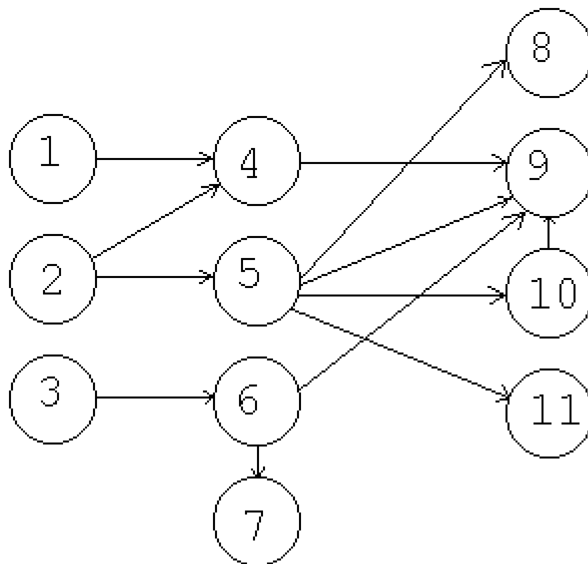
Trata-se de unha lista de listas. A lista principal ten unha entrada por cada vértice (|V|) e cada posición da lista principal, cada vértice v, ten unha lista indicando todas os vértices aos cais hai arista desde v a eles. Como entre todas as sublistas haberá tantas posicións como aristas totais |A| haxa no grafo, agora o requerimento de espazo será O(|V|+|A|). Nos grafos non dirixidos, cada arista aparece duas veces, unha vez nunha lista e outra vez en outra, asi que o requerimento de espazo será o dobre para esta representación.

Para atopar todos os vértices adxacentes a un dado hai que percorrer a sublista.

Para o problema de asignar números a vértices que tên nomes alfanuméricos non procesábeis, a solución mais sinxela é empregar unha tábua de dispersión, na cal se armacene un nome e un número interno no intervalo 1 a |V| para cada vértice.

4.2.- Ordenación topolóxica.

A ordenación topolóxica é unha ordenación dos vértices de un grafo dirixido acíclico, tal que se hai un camiño de v a w, entón w aparece despois de v na ordenación topolóxica.



Este grafo representa unha série de asignaturas necesárias para conseguir unha titulación. Cando hai una arista de unha asignatura a outra quer dicer que a primeira debe ser aprobada para se poder matricular da segunda. Note-se por que se requere que o grafo sexa acíclico: se tivese ciclos non teria sentido a ordenación (unha asignatura precederia a outra e esta precederia à primeira, contradicción).

A ordenación topolóxica non é única. No exemplo vemos como hai muitos xeitos distintos para conseguir a titulación. Por exemplo:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

2, 3, 6, 5, 1, 4, 11, 8, 10, 7, 9

Definimos grau de entrada de un vértice como o número de aristas que chegan a ese vértice.

ALGORITMO DE ORDENACIÓN TOPOLÓXICA

precondicións:
- grafo foi lido
- usamos listas de adxacência para representar o grafo
- obtívo-se a lista de graus de entrada de cada vértice
  (a partir do grafo)

postcondicións:
- obtención do vector numero_topolóxico

proceso:

1º) Buscar vértice sen número de ordenación (ainda non foi procesado, non ten asignada orden na clasificación topolóxica) e que teña grau de entrada cero.

2º) Asignar-lle a este vértice o número de ordenación topolóxica correspondente ao paso actual (seguindo os naturais: 1, 2, ... é un simples contador que se actualiza en cada execución do bucle).

3º) Borrar todas as aristas que saen do vértice escollido.

4º) Se quedan vértices por asignar-lles número de orden topolóxica, voltar ao paso 1º), senón rematar.

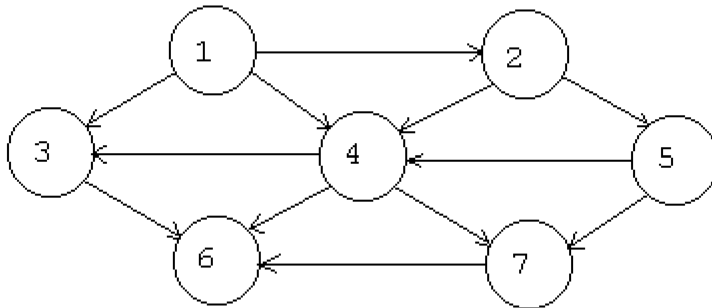
```
{-------------------------------------------------------------}
procedimento ordenación_topolóxica ( G : tipoGrafo )
      contador ← 1
      erro ← falso
      mentres contador ≤ |V| { procesar todos os vértices }
      e non erro { non atopamos ciclos }
      facer
            v ← buscar_novo_vértice_grau_ent_cero
                                                ( lista_graus_ent )
                  { os nodos están numerados de 1 a n }
                  { aqui devolve-se 0 se non hai nodo de grau ent 0 }
            se v = 0
            entón
                  erro ← certo
                  erro ( "o grafo ten un ciclo" )
            senón
                  número_topolóxico [ v ] ← contador
                  para cada w adxacente a v facer
                        grau_ent [ w ] ← grau_ent [ w ] - 1
                              { borramos arista v → w }
                  finpara
                  contador ← contador + 1
            finse
      finmentres
```

```
finprocedimento
{-----------------------------------------------------------------}
```

A función buscar_novo_vértice_grau_ent_cero será determinante na eficiência. Busca un vértice de grau de entrada cero que ainda non fose procesado (ao que ainda non se lle asignou un número de orden topolóxica).

Cando buscar_novo_vértice_grau_ent_cero é un simples percorrido secuencial do arranxo grau_ent cada chamada à función leva un tempo $O(|V|)$. Como isto o hai que facer no bucle principal do procedimento para cada vértice, o tempo de execución do algoritmo é $O(|V|^2)$.

Pode-se mellorar isto criando unha estrutura separada contendo os vértices de grau de entrada cero e que ademais ainda non foron asignados. Neste caso buscar_novo_vértice_grau_ent_cero devolverá (e elimina da mesma) o primeiro elemento de esta estrutura, e o algoritmo xá non será $O(n^2)$ senón que será $O(|A|+|V|)$. Cando decrementamos os graus de entrada dos vertices adxacentes, revisamos cada vértice, e colocamo-lo nesta estrutura auxiliar se o seu grau pasou agora a ser cero. A estrutura auxiliar é mellor implementá-la como unha cola.



vértice	grau de entrada "–" = xá escollido en negriña os que câmbian
1	0 – – – – – –
2	1 **0** – – – – –
3	2 **1** 1 1 **0 – –**
4	3 **2 1 0** – – –
5	1 1 **0** – – – –
6	3 3 3 3 **2 1 0**
7	1 1 1 **0** 0 0 –
saída*	1,2,5,4, 3, 7,6
cola	[1][2][5][4,7][3,7][7][]

*ordenación topolóxica

```
{-----------------------------------------------------------------}
procedimento ordenación_topolóxica_2 ( G : tipoGrafo )
      criar_cola ( grau_cero_e_sen_asignar )
      para cada vértice v facer
            se grau_ent [ v ] = 0
            entón insertar_cola ( grau_cero_e_sen_asignar , v )
      finpara
      contador ← 1
      mentres non cola_valeira ( grau_cero_e_sen_asignar ) facer
```

```
            v ← eliminar_cola ( grau_cero_e_sen_asignar )
            número_topolóxico [ v ] ← contador
            contador ← contador + 1
            para cada w adxacente a v facer
                grau_ent [ w ] ← grau_ent [ w ] - 1
                se grau_ent [ w ] = 0
                entón insertar_cola ( grau_cero_e_sen_asignar , w )
                finse
            finpara
      finmentres
      se contador ≤ |V|
      entón erro ( "o grafo ten ciclos" )
finprocedimento
{----------------------------------------------------------------}
```

Pasamos de $O(n^2)$ a $O(|A|+|V|)$, xá que o proceso fai-se percorrendo todos os vértices ($|V|$) e en cada vértice vai-se acceder às aristas que saen del, en conxunto $|A|$ para todos os vértices.

4.3.- Árbore de expansión mínima (árbore expandida).

Dado un grafo G=(V,A) *conexo, non dirixido e pesado* (custo de todas as aristas ≥ 0), trata-se de atopar un grafo G'=(V,T) onde T é un subconxunto de A tal que:

-todos os vértices sigan conectados
-o sumatório dos custos de todas as aristas é mínimo

$|T| = |V|-1$ é o número mínimo de aristas que pode ter T

Se $|T| = |V|-1$ entón G' non ten nengún ciclo.

Se $|T| > |V|-1$ entón G' ten polo menos un ciclo. Entón podemos quitar unha arista sen desconectar o grafo. Quitamos unha arista do ciclo e o grafo segue sendo conexo, daquela o sumatório dos custos das aristas ou ben diminui ou ben segue igual (se o peso da arista borrada é cero). Polo tanto $|T| = |V|-1$ é o grafo expandido mínimo e ademais posto que G' é conexo, G' é unha árbore, é dicer que temos a árbore expandida de peso mínimo.

Exemplo de aplicación: Quer-se comunicar por cabo un concello con fibra óptica. O custo das aristas ven dado polo prezo por metro do cabo entre dous pontos (nodos). Trata-se de dar coa instalación mais barata. Isto é un problema de árbore expandida mínima.

Para atopar a árbore expandida mínima empregaremos algoritmos voraces (*greedy algorithms*), que son os que se usan nos problemas de optimización. Trata-se de atopar un conxunto de candidatos en función do obxectivo.

Algoritmo voraz xeral:

1º) inicialmente o conxunto de candidatos é valeiro
2º) en cada paso temos unha función de selección (escoller o mellor de entre todos os candidatos)
3º) o conxunto de candidatos é completábel se admite o candidato; é non completábel se se rexeita o candidato.
4º) Controla-se unha condición a cal indica se o conxunto que se ten é a solución definitiva ou se ainda hai que repetir o algoritmo.

No caso da árbore expandida mínima a condición de terminación é que o conxunto de aristas sexa árbore expandida. É completábel se non hai ciclos.

Def.- Conxunto de aristas é prometedor se *pode ser ampliado* para formar unha solución óptima ao problema.

Def.- Unha arista parte (orixen) de un conxunto de nodos ⇔ unha e só unha das suas extremidades está no conxunto.

LEMA

Hipóteses

G=(V,A) un grafo conexo non dirixido e ponderado
$B \subset V$
v a arista mais curta (menos pesada) que parte de B
$T \subseteq A$
T = {aristas} prometedor sen aristas que partan de B

Entón

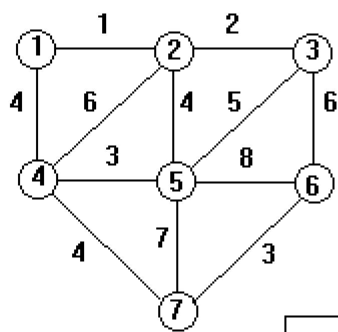
$T \cup \{v\}$ é prometedor

4.3.1.- Algoritmo de Kruskal.

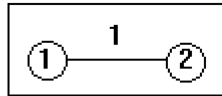
Inicialmente T = {} e engadimos aristas en T de tal maneira que durante todo o desenrolo (N,T) é un conxunto de componentes conexas. Ao final só queda unha componente conexa que é a árbore expandida de peso mínimo.

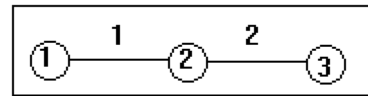
Como engadir unha arista? cal é a función de selección? Toma-se a arista mais curta que una nodos de componentes conexas distintas (o cal equival a dicer que non se produza un ciclo por engadir esa arista).

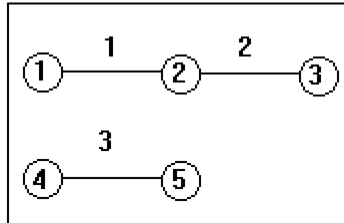
Exemplo:

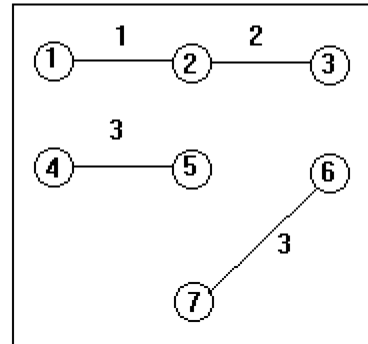


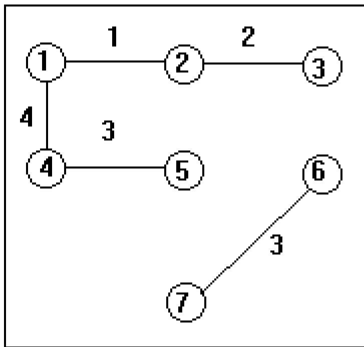
Kruskal: engadir aristas de peso mínimo sen formar ciclos

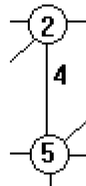




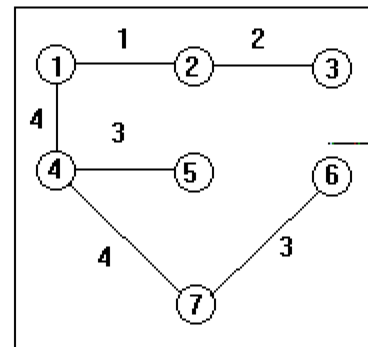








rexeitado por causar un ciclo



Ordenación de aristas de menor a maior peso:

lonxitude	aristas
1	{1,2}
2	{2,3}
3	{4,5} {6,7}
4	{1,4} {2,5} {4,7}
5	{3,5}
6	{2,4} {3,6}
7	{5,7}
8	{5,6}

paso	arista	componentes conexas
0	-	{1}{2}{3}{4}{5}{6}{7}
1	{1,2}	{1,2}{3}{4}{5}{6}{7}
2	{2,3}	{1,2,3}{4}{5}{6}{7}
3	{4,5}	{1,2,3}{4,5}{6}{7}
4	{6,7}	{1,2,3}{4,5}{6,7}
5	{1,4}	{1,2,3,4,5}{6,7}
6	{2,5}	{1,2,3,4,5}{6,7} (cria ciclo)
7	{4,7}	{1,2,3,4,5,6,7}

Teorema.-
O algoritmo de Kruskal calcula a árbore expandida de peso mínimo.

Implementación:
-lista de componentes conexas
-operacións:

- buscar (x) = averiguar componente onde está o nodo x
- fusionar (A , B) = converter 2 componentes conexas nunha soa

Han de implementar-se eficientemente porque se usan muitas veces

T.D.A. CONXUNTO DISXUNTO (para implementar as operacións)

1ª representación

Menor valor como representante.
T[1..n] onde T[i] é o representante do conxunto ao que pertence i

Exemplo: representar os seguintes conxuntos disxuntos
{1,5} mediante o 1
{2,4,7,10} mediante o 2
{3,6,8,9} mediante o 3

1	2	3	2	1	3	2	3	3	2

```
{-------------------}
FUNCIÓN BUSCAR1 ( x )
{busca a que conxunto pertence x}
{devolve o representante do conxunto ao que pertence x}
      DEVOLVER T [ x ]
FINFUNCIÓN
{-----------------------------}
PROCEDIMENTO FUSIONAR1 ( a , b )
{recebe representantes de cada un dos dous conxuntos}
{actualiza o representante dos elementos dos dous conxuntos representados
polos elementos a e b}
      I ← MIN ( a , b )
      J ← MAX ( a , b )
      PARA K ← 1 ATÉ N FACER
            SE T [ K ] = J
            ENTÓN T [ K ] ← I
            FINSE
      FINPARA
FINPROCEDIMENTO
```

Secuência de n operacións:

buscar: até n veces $\rightarrow O(1)\cdot n=O(n)=\theta(n)$

fusionar: até N-1 veces $\rightarrow \theta(N)\cdot(N-1)=\theta(N^2)$

$\Big\}\ \theta(n^2)$, $n \sim N$

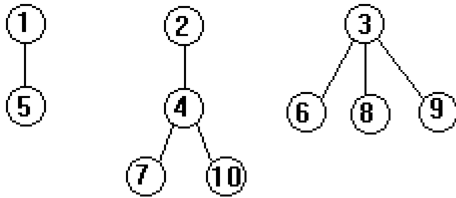
2ª representación (simplifica o fusionar)

Temos unha árbore por cada conxunto.

T[i] = i → T[i] é o representante do conxunto (escolle-se o menor elemento como representante do conxunto)

T[i] ≠ i → T[i] é o pai de i na árbore correspondente

Exemplo:



representados mediante ({1,5} {2,4,7,10} {3,6,8,9}):

1	2	3	2	1	3	4	3	3	4

```
{-------------------}
FUNCIÓN BUSCAR2 ( x )
{busca a que conxunto pertence x}
{devolve o representante do conxunto ao que pertence x}
      r ← x
      MENTRES T [ r ] ≠ r
      FACER r ← T [ r ]
      FINMENTRES {chegar à raiz}
      DEVOLVER r
FINFUNCIÓN
{-----------------------------}
PROCEDIMENTO FUSIONAR2 ( a , b )
{recebe representantes de cada un dos dous conxuntos}
{actualiza o representante dos elementos dos dous conxuntos representados
polos elementos a e b}
      SE a < b
      ENTÓN T [ b ] ← a
      SENÓN T [ a ] ← b
      FINSE
FINPROCEDIMENTO
```

Buscar2 é Θ(N) no pior caso (o pior caso é que só teñamos un conxunto, xá que están todos os elementos nunha soa coluna).

Fusionar2 é O(1)

n Buscar2 e N-1 Fusionar2 é Θ(nN) no pior caso
como n~N o pior caso é $\Theta(n^2)$

O remédio é limitar a altura das árbores para limitar o tempo que leva o percorrido desde unha folla até a raiz da árbore.

En Fusionar: trata-se de que a árbore mais baixa se convirta no fillo da árbore mais alta [h_a=altura do conxunto cuio representante é a]

$$\text{Fusionar(a,b) ten altura} \begin{cases} \max(h1,h2) & \text{se } h1 \neq h2 \\ h1+1 & \text{se } h1 = h2 \end{cases}$$

Teorema

Despois de unha secuência suficientemente longa, unha árbore de k nodos terá unha altura máxima de $\lfloor \log k \rfloor$

A demostración fai-se por indución sobre o número de nodos.

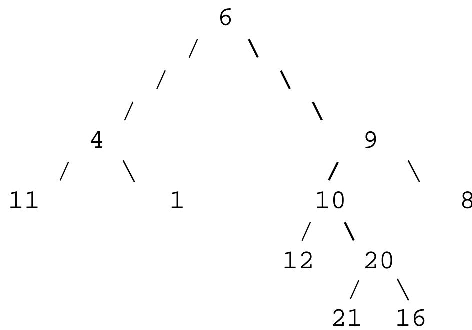
Implementación: vector adicional A para ter as alturas das árbores de tamaño n para n conxuntos.

```
{-----------------------------}
PROCEDIMENTO FUSIONAR3 ( a , b )
{recebe representantes de cada un dos dous conxuntos}
{actualiza o representante dos elementos dos dous conxuntos representados polos elementos a e b}
      SE A[a] = A[b]
      ENTÓN
            A[a] ← A[a] +1
            T[b] ← a
      SENÓN
            SE A[a] > A[b]
            ENTÓN T[b] ← a
            SENÓN T[a] ← b
            FINSE
      FINSE
FINPROCEDIMENTO
```

n Buscar e N-1 Fusionar3 é Θ(N+n·log N) no pior caso
como n~N o pior caso é Θ(n·log n)

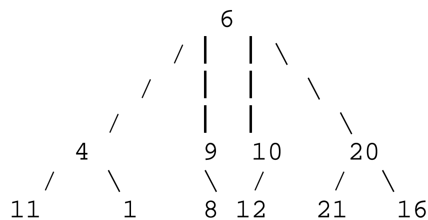
Técnica de compresión de camiños: buscar mellora.

Exemplo:



Executa-se Buscar3 (20)

Os elementos intermédios converten-se en fillos da raiz:



Isto só compensa facê-lo se hai que realizar muitas buscas. Con muitas buscas: todas as follas acaban por ser fillos da raiz, entón Buscar3 é O(1).

A fusión perturba temporalmente este estado en que todas as follas son fillos da raiz.

A maior parte do tempo, Buscar3 e Fusionar3 executan-se nun tempo constante O(1) co cal obtemos tempo O(n) para unha secuência de n operacións (para un caso de n~N, n e N da mesma orden).

Hai un problema: A[a] xá non indica a altura de a, pero podemos supor que A[a] da un límite superior, unha cota para a altura de a, co cal é suficiente para realizar fusións.

```
función buscar ( x )
      r ← x
      mentres T [ r ] ≠ r
      facer r ← T [ r ]
      finmentres
      i ← x
      mentres i ≠ r
      facer
            j ← T [ i ]
            T [ i ] ← i
            i ← j
      finmentres
      devolver r
finfunción
```

Voltamos agora ao algoritmo de Kruskal que fora o problema que orixinara esta estrutura de dados que vimos de ver.

```
{----------------------------------------------------------------}
función Kruskal ( G = ( N , A ) )
{código de inicialización:}
   ordenar A segundo lonxitudes crescentes
   n ← |N|
   T ← ∅ {conxunto de aristas da árbore que estamos calculando}
   inicializar os n conxuntos,
   cada nodo estará nun conxunto disxunto

   repetir
      a ← {u,v} {arista mais curta de A áinda sen considerar}
      comp_u ← buscar ( u )
      comp_v ← buscar ( v )
      se comp_u ≠ comp_v
      entón
            fusionar ( comp_u , comp_v )
            T ← T ∪ {a}
      finse
   até que o número de elementos que hai en T sexa igual a n-1
   {|T|=n-1}
   devolver T
finfunción
{----------------------------------------------------------------}
```

n nodos, m aristas (en G):

- $\Theta(m \cdot \log m)$ ordenar A

$\equiv \Theta(m \cdot \log n)$ xá que podemos acotar m

$n-1 \le m \le n(n-1)/2$

$n(n-1)/2$ nº máximo de aristas en grafo completo

- $\Theta(n)$ para inicializar n conxuntos disxuntos
- $\Theta(2m \cdot \log n)$ 2m Buscar e n-1 Fusionar
- $O(m)$ para o resto no pior caso

A suma de todos estes termos da que Kruskal executa-se nun tempo $\Theta(m \cdot \log n)$

Melloras (non câmbian o pior caso): inicialización en $\Theta(m)$, busca de arista mínima en $\Theta(\log m)=\Theta(\log n)$.

4.3.2.- Algoritmo de Prim.

Sexa o grafo a procesar G=(N,A)

Sexa B = {nodos}
T = {aristas}

Inicialmente B = {nodo arbitrário de N}
T = ∅

Cada paso: escoller {u,v} mais curta tal que $u \in B$
$v \in N\backslash B$

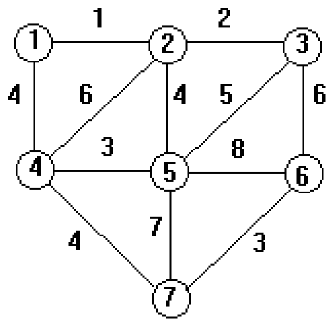
As aristas de T forman en todo momento a árbore extendida mínima para os nodos de B.

Engadir v a B
{u,v} a T

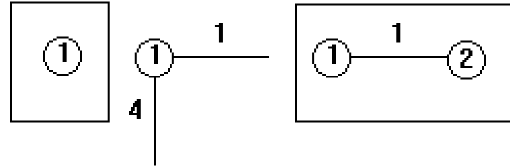
Repetir o proceso até que B = N

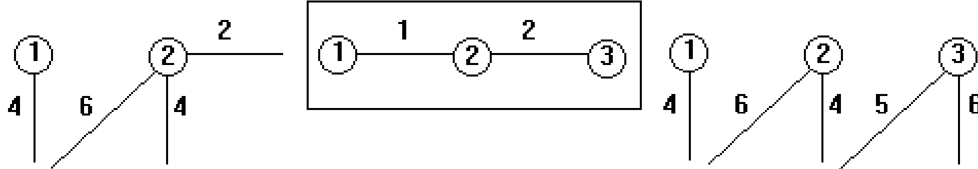
```
{-------------------------------------------------------------}
función Prim ( G = ( N , A ) )
      T ← ∅
      B ← {un nodo de N}
      mentres B ≠ N facer
            a ← {u,v} arista mais curta / (u∈B ∧ v∈N\B)
            T ← T ∪ { a }
            B ← B ∪ { v }
      finmentres
      devolver T
finfunción
{-------------------------------------------------------------}
```

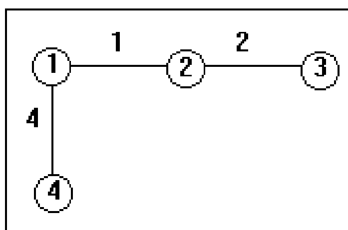
Exemplo:

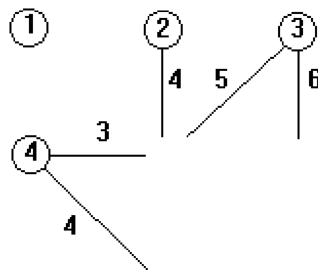


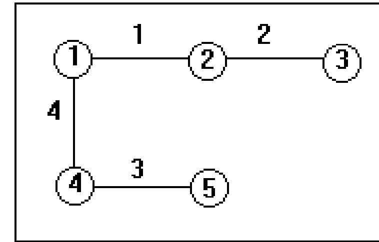
Prim: escoller un nodo inicial. Tomar arista de peso mínimo entre conxunto escollido e exterior. Actualizar conxunto escollido. Repetir até que todos nodos conectados.

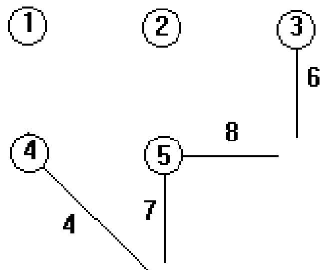


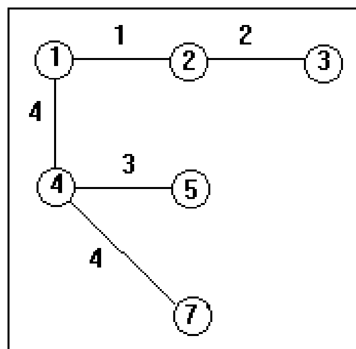


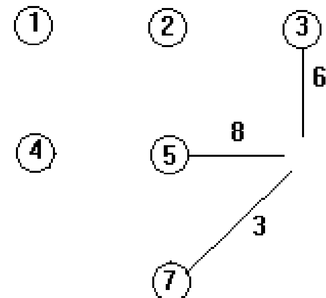


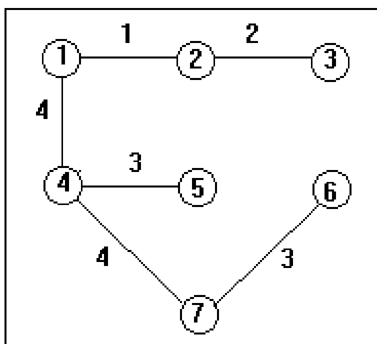












paso	a	B
0	-	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}

4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	N

Teorema

O algoritmo de Prim calcula a árbore extendida mínima.

Demostración por indución sobre o número de aristas de T.

É prometedor en cada paso → solución óptima.

Base: T={} é prometedor.

Indución: T é prometedor antes de engadir a={u,v} → B ⊂ N

T é prometedor sen aristas que partan de B
a é unha das aristas mais curtas que parten de B
polo Lema
T∪{a} tamén é prometedor

Implementación sinxela

N = {1,2,...,n}

L matriz simétrica / L(i,j)=distância
L(i,j)=∞ se non existe arista {i,j}

∀ nodo i ∈ N\B, mais_próximo [i] = nodo mais próximo a i
dist_mínima [i] = distância de i a mais_próximo [i]

Se i ∈ B entón dist_mínima [i] = -1

```
{-----------------------------------------------------------------}
función Prim ( L [ 1 .. n , 1 .. n ] )
      T ← ∅ {inicialmente dist_min [ i ] = -1}
      para i ← 2 até n facer
            mais_próximo [ i ] ← 1
            dist_min [ i ] ← [ i , 1 ]
      finpara
      {bucle voraz}
      repetir n-1 veces
            min ← ∞
            para j ← 2 até n facer
                  se 0 ≤ dist_min [ j ] < min
                  entón
                        min  ← dist_min [ j ]
                        k ← j
                  finse
            finpara
            T ← T ∪ { mais_próximo [ k ] , k }
            dist_min [ k ] ← -1 {engadir k en B}
            para j ← 2 até n facer
                  se L [ j , k ] < dist_min [ j ]
```

```
               entón dist_min [ j ] ← L [ j , k ]
               finse
          finpara
     finrepetir
     devolver T
finfunción
{----------------------------------------------------------------}
```

Bucle voraz: n-1 veces, cada paso é O(n) → Prim é $O(n^2)$

Comparación: Kruskal é O(m·log n) con m=|A|

- grafo denso: m → n(n-1)/2 → Kruskal en $O(n^2 \cdot \log n)$ pior que Prim
- grafo disperso: m → n → Kruskal en O(n·log n) mellor que Prim

Poden-se usar montículos para implementar Prim, entón resulta en O(m·log n)

Existen algoritmos mais eficientes para o cálculo de árbores expandidas mínimas.

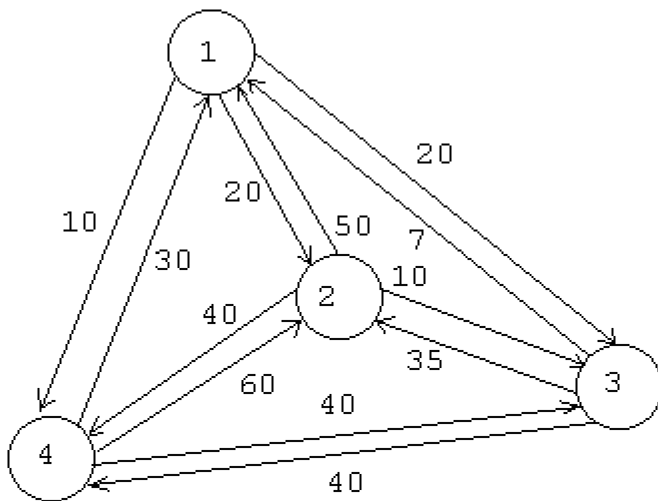
4.4. Algoritmo do camiño mais curto (Dijkstra)

Sexa G = (N , A) un grafo dirixido, con lonxitudes de aristas ≥ 0

Consideramos un nodo a orixen dos camiños. O problema que se plantexa é averiguar a lonxitude do camiño mais curto da orixen a todos os restantes nodos de G.

O algoritmo voraz que resolve isto é o algoritmo de Dijkstra. Traballamos con dous conxuntos de nodos S e C. S son os nodos escollidos, coñece-se a distância mínima de cada un de eles à orixen. C é o conxunto de nodos candidatos. A invariante do algoritmo é que N = S ∪ C. Inicialmente temos que S = { orixen }, e finalmente temos que S = N. En cada paso trata-se de escoller o nodo candidato, nodo de C, tal que a sua distância desde a orixen a él é a menor posíbel. Dicimos que un nodo é especial se todos os seus nodos intermédios pertencen a S. Manexamos un vector D contendo en cada posición i a lonxitude do camiño especial mais curto ao nodo i-ésimo do grafo. Ao engadir un nodo v en S, o camiño especial mais curto a v é tamén o camiño menor de todos os camiños a v. Ao final temos todos os nodos en S, todos os camiños entre a orixen e un nodo calquer son especiais, entón en D temos a solución.

Exemplo 1:



-1	20	20	10
50	-1	10	40
7	35	-1	40
10	60	40	-1

Tomamos como nodo inicial o "1"

Evolución do vector D:
A marca _ por arriba indica que o nodo está escollido, en S
(senón é candidato, está en C)
A indicación ^^ por abaixo indica que se escolle a arista menor das que levan a nodos ainda non escollidos.

```
 _
[-1|20|20|10] inicialmente coluna [nodo orixen,i] ∀i
         ^^

 _       __
[-1|20|20|10]de 1 a 2 pode-se ir polo camiño actual: 20
                 de 1 a 2 pode-se ir desde 4: (10+) 60
                       queda 20 que é o actual, é menor
                 de 1 a 3 pode-se ir polo camiño actual: 20
                 de 1 a 3 pode-se ir desde 4: (10+) 40
                       queda 20 que é o actual, é menor

 _ __    __
[-1|20|20|10]de 1 a 3 pode-se ir polo camiño actual: 20
   ^^       de 1 a 3 pode-se ir desde 2: (20+) 20
                       queda 20 que é o actual, é menor
```

Entón a solución é 20, 20, 10 como camiños mais curtos do nodo 1 ao 2, 3, e 4 respeitivamente.

[onde pon "camiño actual" entenden-se "camiño especial menor atopado até agora"]

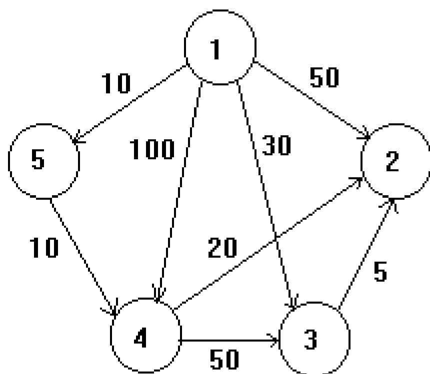
Tomamos como nodo inicial o "3"

```
D [ 7|35|-1|40]
    ^
                  de 3 a 2 pode-se ir polo camiño actual: 35
                  de 3 a 2 pode-se ir desde 1: (7+) 20
                        substitución de 35 por 27 por ser 27 menor

D [ 7|27|-1|40]
     ^      de 3 a 4 pode-se ir polo camiño actual: 40
                  de 3 a 4 pode-se ir desde 1: (7+) 10
                        substitución de 40 por 17 por ser 17 menor

D [ 7|27|-1|17]
            ^^    de 3 a 2 pode-se ir polo camiño actual: 27
                  de 3 a 2 pode-se ir desde 4: (17+) 60
                        queda 27 que é o actual, é menor
```

Entón a solución é 7, 27, 17 como camiños mais curtos do nodo 3 ao 1, 2, e 4 respeitivamente.

```
{-----------------------------------------------------------------}
función Dijkstra ( L [ 1 .. n , 1 .. n ] )
      C ← { 2 .. n } {o nodo inicial é o 1, xá non é candidato}
      para i ← 2 até n facer
            D [ i ] ← L [ 1 , i ] {copiar fila de nodo inicial}
      finpara
      {bucle voraz}
      repetir n-2 veces
            v ← nodo de C que minimiza D [ v ]
                  {nodo non escollido ainda con arista menos pesada}
            C ← C \ { v }
            para cada w ∈ C facer
                  D [ w ] ← min ( D [ w ] , D [ v ] + L [ v , w ] )
            finpara
      finrepetir
      devolver D
finfunción
{-----------------------------------------------------------------}
```

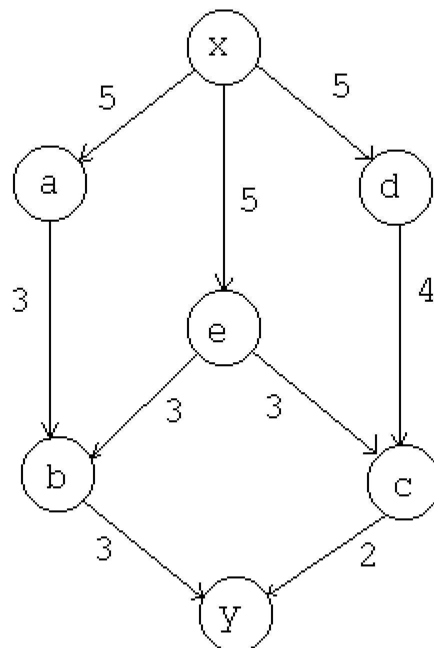
Exemplo 2:



-1	50	30	100	10
-1	-1	-1	-1	-1
-1	5	-1	-1	-1
-1	20	50	-1	-1
-1	-1	-1	10	-1

paso	v	C candidatos	D solución
ini	-	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,*20*,10]
2	4	{2,3}	[40,*30*,20,10]
3	3	{2}	[35,30,20,10]

Exemplo 3:



	x	a	b	c	d	e	y
x	-1	5	-1	-1	5	5	-1
a	-1	-1	3	-1	-1	-1	-1
b	-1	-1	-1	-1	-1	-1	3
c	-1	-1	-1	-1	-1	-1	2
d	-1	-1	-1	4	-1	-1	-1
e	-1	-1	3	3	-1	-1	-1
y	-1	-1	-1	-1	-1	-1	-1

-1 representa infinito (observar como operamos con -1)
subliñados nodos escollidos (S)
en negriña referência actual

	x	**a**	b	c	d	e	y
D	-1	5	-1	-1	5	5	-1

a→b= 3 +5= 8 vs (→b actual = -1) → b: 8
a→c=-1 +5= -1 vs (→c actual = -1) → c: -1

a→d=-1 +5= -1 vs (→d actual = 5) → d: 5
a→e=-1 +5= -1 vs (→e actual = 5) → e: 5
a→y=-1 +5= -1 vs (→y actual = -1) → y: -1

	x	a	b	c	**d**	e	y
D	-1	5	8	-1	5	5	-1

d→b=-1 +5= -1 vs (→b actual = 8) → b: 8
d→c= 4 +5= 9 vs (→c actual = -1) → c: 9
d→e=-1 +5= -1 vs (→e actual = 5) → e: 5
d→y=-1 +5= -1 vs (→y actual = -1) → y: -1

	x	a	b	c	d	**e**	y
D	-1	5	8	9	5	5	-1

e→b= 3 +5= 8 vs (→b actual = 8) → b: 8
e→c= 3 +5= 8 vs (→c actual = 9) → c: 8
e→y=-1 +5= -1 vs (→y actual = -1) → y: -1

	x	a	**b**	c	d	e	y
D	-1	5	8	8	5	5	-1

b→c=-1 +8= -1 vs (→c actual = 8) → c: 8
b→y= 3 +8= 11 vs (→y actual = -1) → y: 11

	x	a	b	**c**	d	e	y
D	-1	5	8	8	5	5	11

c→y= 2 +8= 10 vs (→y actual = 11) → y: 10

	x	a	b	c	d	e	y
D	-1	5	8	8	5	5	10

Unha solución mais detallada consiste en mirar o conxunto de nodos polos que pasa o camiño mais curto. Modificacións a facer:

Incluir P[2..n] onde P[v]=nodo que precede a v no camiño mais alto. Para achar o camiño mais alto hai que seguir os precedentes até a orixen.

No algoritmo:

- Engadir ao comenzo P[i] ← 1 ∀i=1,...,n
- para cada w ∈ C facer

```
      se D[w] > D[v] + L[v,w]
      entón
            D[w] ← D[v] + L[v,w]
            P[w] ← v
      finse
  finpara
```

Teorema

Dijkstra atopa os camiños mais curtos desde un nodo orixen a todos os outros nodos do grafo.

Demostración

Por indución, demostrar:

a) se nodo $i \neq 1$ está en S

D[i] = lonxitude do camiño mais curto $1 \rightarrow i$

b) se nodo i non está en S, está en C

D[i] = lonxitude do camiño especial mais curto $1 \rightarrow i$

[consultar bibliografia]

Análise do algoritmo de Dijkstra

$|N|=n$
$|A|=m$
L[1..n,1..n]
Inicialización: O(n)
Escoller un nodo v que minimiza D

→ percorrer todos os elementos de C

n-1, n-2, ..., 2 valores en D → $\Theta(n^2)$

"Para" aniñado do algortimo

n-2, n-3, ..., 1 iteracións = $\Theta(n^2)$

Entón **Dijkstra é $\Theta(n^2)$**

Caso de grafo pouco denso é dicer que $m<<<n^2$

-usaremos listas de adxacência: aforramos no "para" aniñado un tempo no acceso a L

-Evitar $\Omega(n^2)$ para resolver v?

Usar montículo para implementar C ordenado segundo D[v]

- Inicialización montículo en $\Theta(n)$
- Extrair $C \leftarrow C \setminus \{v\}$ en O(log n)
- "Para" aniñado → modificar D[w] en O(log n)
  isto fai-se como máximo unha vez por arista

Extrair raiz do montículo n-2 veces e modificamos un máximo de m veces un valor de D

Σ → Dijkstra é $\Theta((m+n) \cdot \log n)$

A implementación rápida é aconsellábel se o grafo é denso.

<fin do tema 4>