

# Árboles abarcadores de costo mínimo

Pablo R. Ramis

Universidad Nacional de Rosario, Instituto Politécnico, Dto. de Informática,  
prramis@ips.edu.ar,  
WWW home page: <http://informatica.ips.edu.ar>

**Resumen** Si bien todos los conceptos necesarios han sido estudiados dentro de Teoría de Grafos, repasaremos algunos que son imprescindibles para el tema que nos toca.

Un grafo no dirigido  $G = V, A$  consta de un conjunto finito de vértices  $V$  y de un conjunto de aristas  $A$ . Se diferencia de un grafo dirigido es que cada arista en  $A$  es un par no ordenado de vértices. Si  $(v, w)$  es arista no dirigida, entonces  $(v, w) = (w, v)$ .

Algunas definiciones útiles:

1. Un camino es una secuencia de vértices

$$v_1, v_2, \dots, v_n$$

tal que

$$(v_i, v_{i+1})$$

es una arista para

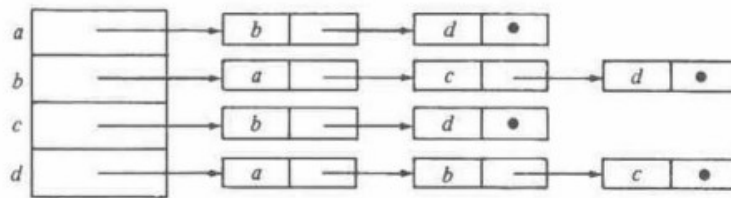
$$1 \leq i \leq n$$

2. Un camino es *simple* si todos sus vértices son distintos con excepción de  $v_1$  y  $v_n$ .
3. La longitud de un camino es  $n - 1$  que es la cantidad de aristas que lo componen.
4. Se dice que un camino  $v_1, v_2, \dots, v_n$  conecta a  $v_1$  con  $v_n$ .
5. Un grafo es *conexo* si todos sus pares de vértices están conectados.
6. Sea  $G = V, A$  un grafo con conjunto de vértices  $V$  y conjunto de aristas  $A$ . Un *subgrafo* de  $G$  es un grafo  $G' = V', A'$  donde:
  - $V'$  es un subconjunto de  $V$ .
  - $A'$  consta de las aristas  $(v, w)$  en  $A$  tales que  $v$  y  $w$  están en  $V'$ .
7. Un *ciclo* (simple) de un grafo es un camino (simple) de longitud mayor o igual a 3 que conecta un vértice con sí mismo.
8. Un grafo conexo acíclico se lo conoce como *árbol libre*. Un grafo de estas características puede convertirse en un *árbol ordinario* si se elige un vértice como raíz y se orienta cada arista desde ella.
9. Todo árbol con  $n \geq 1$  vértices contiene  $n - 1$  arista. Si se agregara cualquier arista resulta un ciclo.

Las formas en representar un grafo pueden ser muchas, las más comunes serían a través de una matriz de adyacencia y/o de listas de adyacencias.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	0	1	0	1
<i>b</i>	1	0	1	1
<i>c</i>	0	1	0	1
<i>d</i>	1	1	1	0

**Figura 1.** Representación bajo Matriz de adyacencia



**Figura 2.** Representación mediante listas enlazadas

## 1. Árboles abarcadores de costo mínimo

Suponga que  $G = V, A$  es un grafo conexo en donde cada arista  $(u, v)$  de  $A$  tiene un costo asociado  $c(u, v)$ . Un *árbol abarcador* para  $G$  es un árbol libre que conecta todos los vértices de  $V$ ; su *costo* es la suma de los costos de las aristas del árbol.

### 1.1. Algoritmo de Kruskal

Volvamos a suponer un grafo conexo  $G = V, A$ , con

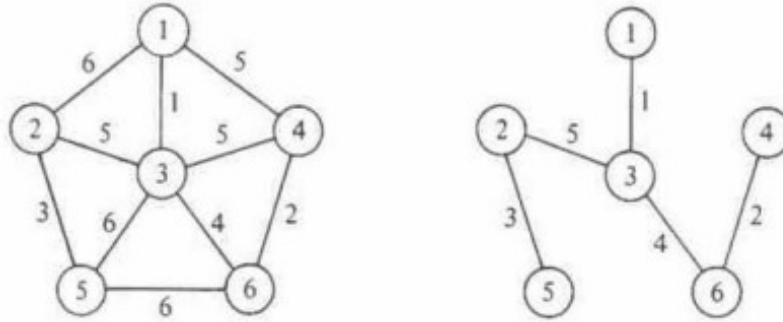
$$V = 1, 2, \dots, n$$

y una función de costo  $c$  definida en las aristas  $A$ . Otra forma de construir un árbol abarcador de costo mínimo para  $G$  es empezar con un grafo

$$T = V, \emptyset$$

constituido con los vértices de  $G$  y sin aristas. Por tanto, cada vértice es un componente conexo por sí mismo. Conforme el algoritmo avanza, habrá siempre una conexión de componentes conexos y para cada componente se seleccionarán las aristas que formen un árbol abarcador.

Para construir componentes cada vez mayores, se examinan las aristas a partir de  $A$ , en orden creciente de acuerdo al costo. Si la arista conecta dos



**Figura 3.** Árbol abarcador de costo mínimo

vértices que se encuentran en dos componentes conexos distintos, entonces se agrega la arista en  $T$ . Se descartará la arista si conecta dos vértices contenidos en el mismo componente ya que esto provocaría un ciclo. Cuando todos los vértices están en un solo componente,  $T$  es un árbol abarcador de costo mínimo para  $G$ .

Las operaciones que se aplican son:

1.  $COMBINA(A, B, C)$  para combinar los componentes  $A$  y  $B$  en  $C$  y llamar al resultado  $A$  o  $B$  arbitrariamente.
2.  $ENCUENTRA(v, C)$  para devolver el nombre del componente de  $C$ , del cual el vértice  $v$  es miembro. Esta operación se utilizará para determinar si los dos vértices de una arista se encuentran en el mismo o en distintos componentes.
3.  $INICIAL(A, v, C)$  para que  $A$  sea el nombre de un componente que pertenece a  $C$  y que inicialmente contiene el vértice  $v$ .

Todas estas operaciones forman parte de una estructura de datos abstracta que llamaremos  $COMBINA\_ENCUENTRA$ .

**Conjunto con operaciones  $COMBINA$  y  $ENCUENTRA$**  En ciertos problemas, se empieza se empieza con una colección de objetos, cada uno de ellos contenido en un conjunto, luego se combinan los conjuntos bajo algún orden dado, y de vez en cuando es necesario preguntar en que conjunto se encuentra algún elemento en particular.

La operación  $COMBINA(A, B, C)$  hace  $C$  igual a la unión de  $B$  y  $A$  bajo el supuesto que tanto  $A$  y  $B$  son disjuntos.  $ENCUENTRA(v)$  es una operación en la que se retorna al conjunto en que pertenece  $v$ .

Para una implementación razonable, se deben restringir los tipos de la estructura o reconocer que este conjunto, el COMBINA-ENCUENTRA, en realidad tiene otros dos tipos como "parámetros": el tipo de los nombres de los conjuntos y el tipo de los miembros de esos conjuntos. En muchas aplicaciones se pueden usar enteros como nombres de conjuntos. Si  $n$  es el número de elementos, también se pueden usar enteros en el intervalo  $[1...n]$  para miembros de los componentes. Para la implantación en cuestión, es importante que el tipo de los miembros de los miembros de los conjuntos sea del tipo subintervalo, porque se desea indizar en un arreglo definido en él. El tipo de los nombres del conjunto no es importante, pues es el tipo de los elementos del arreglo, no de sus índices.

Suponiendo que declaramos componentes de tipo CONJUNTO\_CE con la intención de que *componentes*[ $x$ ] contenga el nombre del conjunto en el cuál se encuentra  $x$ . Esquemáticamente quedaría de esta forma (tener en cuenta que esto se plantea como si tratáramos a un conjunto de elementos, nuestra implementación será algo mas compleja en la estructura necesaria):

```

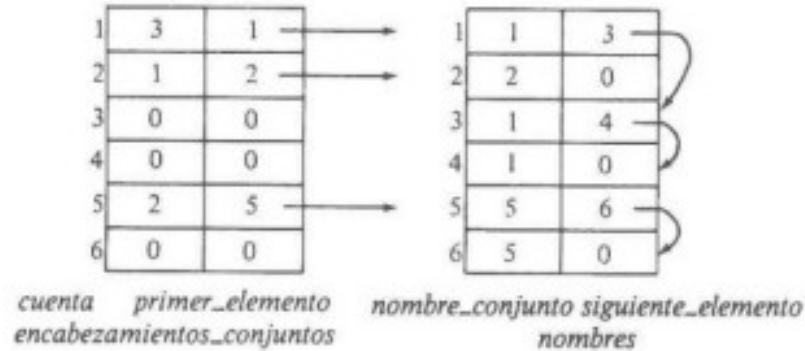
1
2
3  const n = |numero de elementos|;
4
5  type
6      tipo_nombre = 1... n;
7      tipo_elemento = 1... n;
8
9      CONJUNTO_CE = record
10         encabezamientos_conjuntos: array[1... n] of record
11             contador = 0... n;
12             primer_elemento = 0... m;
13         end;
14         nombre: array[1... n] of record
15             nombre_conjunto: tipo_nombre;
16             siguiente_elemento: 0... n;
17         end:
18     end;
19
20 procedure INICIAL (A: tipo_nombre; x: tipo_elemento; var C:
21     CONJUNTO_CE)
22     begin
23         C.nombres[x].nombre_conjunto = A;
24         C.nombres[x].siguiente_elemento = 0;
25         |puntero nulo al siguiente elemento|
26         C.encabezamientos_conjuntos[A].cuenta = 1;
27         C.encabezamientos_conjuntos[A].primer_elemento = x;
28     end; |INICIAL|
29
30 procedure COMBINA (A, B: tipo_nombre; var C: CONJUNTO_CD);
31     var
32         i: 1... n;
```

```

32     begin
33         if C.encabezamientos_conjuntos[A].cuenta >
34             encabezamientos_conjuntos[B].cuenta then
35             begin
36                 |A es el conjunto mas grande, combina B dentro de
37                 A |
38                 |encuentra el final de B, cambiando los nombres
39                 de los conjuntos
40                 por A conforme se avanza|
41                 i := C.encabezamientos_conjuntos[B].
42                 primer_elemento;
43
44                 repeat
45                     C.nombres[i].nombre_conjunto := A;
46                     i := C.nombres[i].siguiente_elemento
47                 until C.nombres[i].siguiente_elemento = 0;
48                 |agrega a la lista A al final de la B y llama A
49                 al resultado |
50                 |ahora i es el índice del último elemento de B |
51
52                 C.nombres[i].nombre_conjunto := A;
53                 C.nombres[i].siguiente_elemento :=
54                 C.encabezamientos_conjunto[A].primer_elemento
55                 ;
56                 C.encabezamientos_conjunto[A].primer_elemento :=
57                 C.encabezamientos_conjunto[B].primer_elemento
58                 ;
59                 C.encabezamientos_conjunto[A].cuenta :=
60                 C.encabezamientos_conjunto[A].cuenta +
61                 C.encabezamientos_conjunto[B].cuenta;
62             end
63         else |B es al menos tan grande como A|
64             |codigo similar al anterior pero intercambiando
65             B por A|
66         end; |COMBINA|
67
68 function ENCUESTRA (x: 1 ... n; var C: CONJUNTO_CE);
69     |devuelve el nombre de aquel conjunto que tiene a x como
70     miembro|
71
72     begin
73         return(C.nombres[x].nombre_conjunto)
74     end; |ENCUESTRA|

```

Podemos ver un ejemplo de la estructura enunciada antes donde el conjunto 1 es 1, 3, 4, el conjunto 2 es 2, y el conjunto 5 es 5, 6



**Figura 4.** CONJUNTO\_CE

Para finalizar veremos el código del algoritmo propiamente dicho. Por una cuestión de trivialidad, no veremos en detalle la implementación de la cola de prioridad ya que no conlleva mayor complejidad.

```

1
2 procedure kruskal (V: CONJUNTO de évrtrices;
3   A: CONJUNTO de aristas;
4   var T: CONJUNTO de aristas)
5
6   var
7     comp_n: integer; |cantidad actual de componentes|
8     aristas: COLA_DE_PRIORIDAD; |conjunto de aristas|
9     componentes: CONJUNTO_CE; |el conjunto V agrupado en
10      un
11      conjunto de componentes COMBINA_ENCUESTRA|
12      U, V: évrtrices;
13      a: arista;
14      comp_siguiete: integer; |nombre para el nuevo
15      componente|
16      comp_u, comp_v; |nombre de los componentes|
17   begin
18     ANULA(T);
19     ANULA(aristas);
20     comp_siguiete := 0;
21     comp_n := únmero de miembros de V;

```

```

21     for v en V do begin |asigna valor inicial a un
22         componente
23         para que contenga un vértice de V|
24         comp_siguiente := comp_siguiente + 1;
25         INICIAL (comp_siguiente, v, componentes)
26     end;
27
28     for a en A do |asigna valor inicial a la cola de
29         prioridad de aristas |
30         INSERTA (a, aristas);
31
32     while comp_n > 1 do begin |considera la siguiente
33         arista|
34         a := SUPRIME_MIN(aristas)
35         sea a = (u, v);
36         comp_u := ENCUENTRA(u, componentes);
37         comp_v := ENCUENTRA(v, componentes);
38
39         if comp_u <> comp_v then begin
40             |a conecta dos componentes diferentes |
41             COMBINA(comp_u, comp_v, componentes);
42             comp_n := comp_n - 1;
43             INSERTA(a, T)
44         end
45     end
46 end; |kruskal|

```

## 1.2. Trabajo Práctico

Planteado la complejidad y estructuras necesarias y teniendo en cuenta los siguientes prototipos. Desarrollar el algoritmo de Kruskal en C. Tener en cuenta que los prototipos son sugeridos.

```

1
2  /*kruskal.h*/
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  typedef int tipo_nombre;
9  typedef int tipo_elemento;
10 typedef int vertice;
11
12 #define VERTICES 5
13
14
15

```

```

16 typedef struct _ARISTA{
17     vertice u;
18     vertice v;
19     int costo;
20 }arista;
21
22 typedef struct _RAMA{
23     struct _ARISTA a;
24     struct _RAMA * sig;
25 }rama;
26
27 typedef struct _ENCABEZADO{
28     int cuenta;
29     int primer_elemento;
30 }encabezado;
31
32 typedef struct _NOMBRE{
33     tipo_nombre nombre_conjunto;
34     int siguiente_elemento;
35 }nombre;
36
37 typedef struct _CONJUNTO_CE{
38     nombre nombres[VERTICES];
39     encabezado encabezamientos_conjunto[VERTICES];
40 }conjunto_CE;
41
42 void inicial(tipo_nombre, tipo_elemento, conjunto_CE*);
43 void combina(tipo_nombre, tipo_nombre, conjunto_CE*);
44 tipo_nombre encuentra(int, conjunto_CE*);
45 void kruskal(rama*);
46
47 void inserta(int, int, int, rama**);
48 arista* sacar_min(rama**);
49 void lista (rama*);

```

```

1
2 /*kruskal.c*/
3
4 #include <kruskal.h>
5
6 int main()
7 {
8
9     int M_Costos[VERTICES][VERTICES];
10    rama *arbol;
11
12    for (int i = 0; i <= VERTICES-1; i++)
13        for (int j = i+1; j <= VERTICES-1; j++)

```



```

14     {
15         printf("Ingrese costo de lado entre vertices %d y
           %d: ",i ,j);
16         scanf("%d",&M_Costos[i][j]);
17     }
18
19     for (int i = 0; i <= VERTICES-1; i++)           //la mitad
inf. de diagonal de matriz
20         for (int j = i+1; j <= VERTICES-1; j++)
21             if (M_Costos [i] [j] != 0)
22                 inserta(i,j,M_Costos[i][j], &arbol); //
inserto en cola prior.
23
24     kruskal (arbol);
25     return 0;
26
27 }

```