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 En este oportunidad y teniendo en cuenta el contexto de nuestra cátedra, saldremos de las implementaciones clásicas y nos obligaremos a encontrar dentro de C++ el modo de utilizar las herramientas que el lenguaje nos brinda. Usaremos en todo lo posible la STL, la potencia de C++ y la programación orientada a objetos.
El algoritmo definido en este apunte se encuentra en el libro Estructuras de datos y algoritmos de Aho, Hopcroft y Ullman. Dicho libro, bibliografía obligatoria sobre el tema, plantea soluciones basadas en estructuras de conjuntos.
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 vertices. Si (v, w) es arista no dirigida, entonces $(v, w) = (w, v)$.
Algunas definiciones útiles:

1. Un camino es una secuencia de vértices

   $$v_1, v_2, ..., v_n$$

   tal que

   $$(v_i, v_i + 1)$$

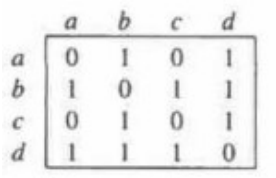
   es una arista para

   $$1 <= i <= n$$

   .
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 $v_1, v_2, ..., v_n$ 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 sigo mismo.

8. Un grafo conexo acíclico se lo conoce como *arbol libre*. Un grafo de estas características puede convertirse en un *árbol ordinario* si se elije un vértice como raíz y se orienta cada arista desde ella.
9. Todo árbol con $n >= 1$ 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.



	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

Figura 1. Representación bajo Matriz de adyacencia

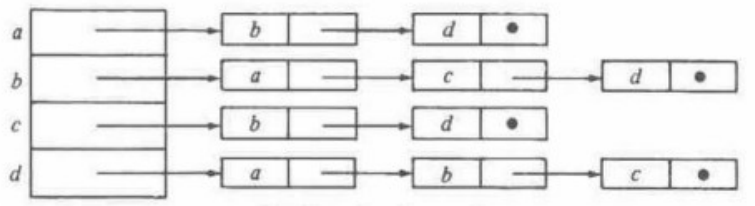


Figura 2. Representación mediante listas enlazadas

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

$$V = 1, 2, ..., n$$

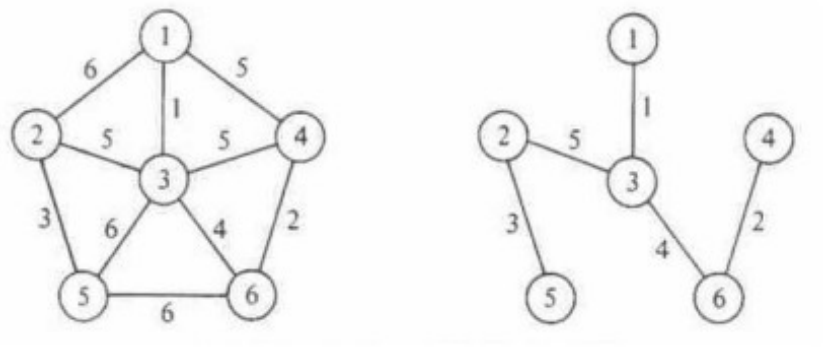


Figura 3. Árbol abarcador de costo mínimo

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 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 dós 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 estructrua 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 del 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 indices.

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 trataramos a un conjunto de elementos, nuestra implementación será algo mas compleja en la estructrua 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             priner_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:
        CONJUNTO_CE)
21       begin
22          C.nombres[x].nombre_conjunto = A;
```

```
23          C.nombres[x].siguiente_elemento = 0;
24              |puntero nulo al siguiente elemento|
25          C.encabezamientos_conjuntos[A].cuenta = 1;
26          C.encabezamientos_conjuntos[A]primer_elemento = x;
27      end; |INICIAL|
28
29  procedure COMBINA (A, B: tipo_nombre; var C: CONJUNTO_CD);
30      var
31          i: 1... n;
32      begin
33          if C.encabezamientos_conjuntos[A].cuenta >
34                  encabezamientos_conjuntos[B].cuenta then
                        begin
35              |A es el conjunto mas grande, combina B dentro de
                     A |
36              |encuentra el final de B, cambiando los nombres
                    de los conjuntos
37                  por A conforme se avanza1
38              i := C.encabezamientos_conjuntos[B].
                    primer_elemento;
39
40              repeat
41                  C.nombres[i].nombre_conjunto := A;
42                  i := C.nombres[i].siguiente_elemento
43              until C.nombres[i].siguiente_elemento = 0;
44              |agrega a la lista A al final de la B y llama A
                    al resultado |
45              |ahora i es el índice del último elemento de B |
46
47              C.nombres[i].nombre_conjunto := A;
48              C.nombres[i].siguiente_elemento :=
49                  C.encabezamientos_conjunto[A].primer_elemento
                        ;
50              C.encabezamientos_conjunto[A].primer_elemento :=
51                  C.encabezamientos_conjunto[B].primer_elemento
                        ;
52              C.encabezamientos_conjunto[A].cuenta :=
53                  C.encabezamientos_conjunto[A].cuenta +
54                      C.encabezamientos_conjunto[B].cuenta;
55          end
56          else |B es al menos tan grande como A|
57               |codigo similar al anterior pero intercambiando
                     B por A|
58
59      end; |COMBINA|
60
61  function ENCUENTRA (x: 1 ... n; var C: CONJUNTO_CE);
62      |devuelve el nombre de aquel conjunto que tiene a x como
            miembro|
63
```

```
64      begin
65          return(C.nombres[x].nombre_conjunto)
66      end; |ENCUENTRA|
```

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 $5, 6$

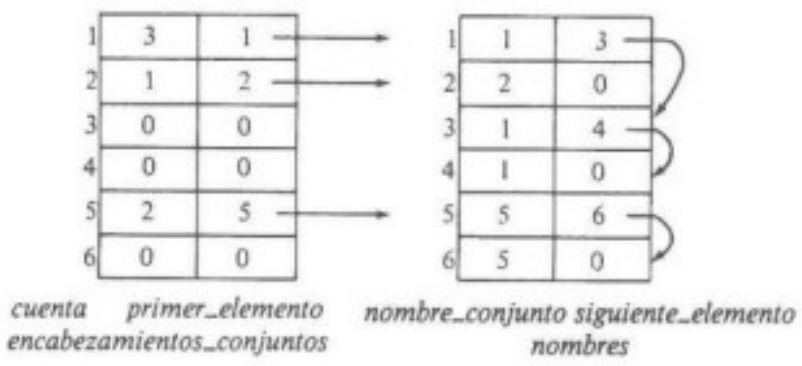


Figura 4. CONJUNTO_CE

Veremos el código del algoritmo propiamente dicho. Como dijimos al principio, las estructuras como la cola de prioridad será la que nos brinda el lenguaje, explicaré brevemente la forma de uso despues de este ejemplo.

```
1
2   procedure kruskal (V: CONJUNTO  de évrtices;
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
                un
10              conjunto de componentes COMBINA_ENCUENTRA|
11          U, V: évrtices;
12          a: arista;
13          comp_siguiente: integer; |nombre para el nuevo
                componente|
```

```
14          comp_u, comp_v; |nombre de los componentes|
15      begin
16          ANULA(T);
17          ANULA(aristas);
18          comp_siguiente := 0;
19          comp_n := únmero de miembros de V;
20
21          for v en V do begin |asigna valor inicial a un
                componente
22              para que contenga un évrtice de V|
23              comp_siguiente := comp_siguiente + 1;
24              INICIAL (comp_siguiente, v, componentes)
25          end;
26
27          for a en A do |asigna valor inicial a la cola de
                prioridad de aristas |
28              INSERTA (a, aristas);
29
30          while comp_n > 1 do begin |considera la siguiente
                arista|
31              a := SUPRIME_MIN(aristas)
32              sea a = (u, v);
33              comp_u := ENCUENTRA(u, componentes);
34              comp_v := ENCUENTRA(v, componentes);
35
36              if comp_u <> comp_v then begin
37                  |a conecta dos componentes diferentes |
38                  COMBINA(comp_u, comp_v, componentes);
39                  comp_n := comp_n - 1;
40                  INSERTA(a, T)
41              end
42          end
43      end; |kruskal|
```

Como saben la STL de C++ brinda muchas estructuras y algoritmos para aplicar sobre ellas.

Algunas de las mismas que recomendamos usar serían:

std::pair Conforman un par de valores que pueden ser diferentes. Cada uno de ellos es accedido através de los miembros públicos *first* y *second*. Su prototipo es sería $template < classT1, classT2 > struct\ pair;$. Admite una estructuras complejas, por ejemplo anidaciones de pares: $pair< pair < char, int >, pair < string, double >>$.

std::map Es un contenedor asociativo, guarda los datos asociados a una key, se ingresan como pares tambien $< key, dato >$ al igual que el pair, admite tipos complejos.

std::priority_queue Es una cola de prioridad, guardará los elementos bajo un criterio específico.

1.2. Trabajo Práctico

Planteado la complejidad y estructuras necesarias y teniendo en cuenta los siguientes prototipos. Desarrollar el algoritmo de Kruskal en C++. Tengan en cuenta el modelado del grafo que ya se ha hecho. El resto de las estructuras declaradas podrían ser cambiadas si lo consideran (siempre que sea usando la librería estandar)

```
1  /**
2   * \type arista
3   * \brief Tipo definido por un pair de pair y entero
4   *
5   * Arista esta compuesta por un pair <pair <char, char>, int>
6   * ambos char son vertice origen y destino y el int el costo
        entre ellos
7   */
8  typedef pair<pair<char, char>,int> arista; /// v1, v2 y costo
       entre ellos
9
10 /**
11  * \struct combina_encuentra
12  * @param nombres
13  * es un map cuya key sera un char (nombre del vertice)
14  * y un pair con el vertice y el vertice siguiente (adyacente
        )
15  * @param encabezados
16  * map cuya key áser un char (vertice) y un pair que tiene a
17  * un intero como indice de cantidad de adyacencias y el
        primer
18  * elemento del conjunto de vertices.
19  */
20 typedef struct combina_encuentra{
21         map<char, pair<char, char>>nombres; ///vertice y
             vertice siguiente
22         map<char, pair<int, char>>encabezados; ///cuenta y
             primer elemento
23 }conjunto_CE;
24
25 class grafo{
26
27         vector<char> V; /// Conjunto de vertices
28         vector<arista> E; /// Conjunto de aristas con sus
             pesos
29
30         conjunto_CE CE; /// Estructura para el analisis de
             adyacencias
31         arbol grafo_ordenado; /// cola de prioridad con los
             pesos de las aristas
32         vector<arista> arbol_minimo; /// Árbol de minimal
```

```
33
34  public:
35          grafo(){} /// constructor
36          ~grafo(){} /// destructor
37
38          void insertar_vertice(const char&); /// Guarda los
                évrtices en el conjunto V
39          void insertar_arista(const char&, const char&, const
                int&);  /// Guarda las aristas en el Conjunto E
40          void insertar_arista(); /// Guarda las aristas en el
                conjunto E pidiendolas ingresar por Teclado
41
42          void inicial (const char&, const char&); ///
                inicializa a las estructuras de conjunto COMBINA-
                ENCUENTRA
43          void combina (const char&, const char&); /// Combina
                las aristas que se encuentran formando el árbol
44          char encuentra (const char&); /// Encuentra los
                vertices dentro del conjunto COMBINA-ENCUENTRA
45          void kruskal (); /// Algoritmo generador del árbol
                recubridor minimal
46          void inserta(); /// Guarda al conjunto E dentro de la
                 cola de prioridad teniendo en cuenta los costos.
47          //arista sacar_min();
48
49          friend ostream& operator <<(ostream&, grafo); ///
                Sobrercarga de la salida áestndar para mostrar al
                 grafo
50  };
```

En la línea 31 vemos al definición del grafo_ordenado el tipo es árbol, dicho código, abajo un poco mas explicado, vemos como se declara una *priority_queue* y se define el método de ordenamiento. Es útil aclarar que para datos simples, por ejemplo *char* la cola ordenaría sin mayor complicación.

Tengamos en cuenta que el prototipo es: $template < classT, classContainer = std :: vector < T >, classCompare = std :: less < typenameContainer :: value_type >> class priority_queue$;

En ejemplo básico:

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4  int main ()
5  {
6      priority_queue <int> cola;
7          cola.push(1);
8          cola.push(3);
9          cola.push(5);
```

```
10          cola.push(7);
11          cola.push(9);
12
13          cout << "ñTamao de la cola (cola.size()): " << cola.
                size() << endl ;
14          cout << "Primer elemento (cola.top()): " << cola.top
                () << endl;
15          cout << "\nContenido : ";
16
17      while (!cola.empty())
18      {
19          cout << '\t' << cola.top();
20          cola.pop();
21      }
22      cout << '\n';
23
24          return 0;
25  }
```

La salida al ejecutar el programa

```
$ g++ -ocola -Wall -std=c++11 cola.cpp
$ ./colañ

Tamao de la cola (cola.size()): 5
Primer elemento (cola.top()): 9

Contenido :     9       7       5       3       1
```

Como vemos, se ordeno con la prioridad del mayor a menor. Si quisieramos cambiar la prioridad la definición de la cola cambia. Con los datos mas simples hariamos de esta forma:

```
1   #include <iostream>
2   #include <queue>
3   using namespace std;
4   int main ()
5   {
6       priority_queue <int, vector<int>, greater<int>> cola;
7           cola.push(1);
8           cola.push(3);
9           cola.push(5);
10          cola.push(7);
11          cola.push(9);
12
13          cout << "ñTamao de la cola (cola.size()): " << cola.
                size() << endl ;
14          cout << "Primer elemento (cola.top()): " << cola.top
                () << endl;
15          cout << "\nContenido : ";
```

```
16
17      while (!cola.empty())
18      {
19          cout << '\t' << cola.top();
20          cola.pop();
21      }
22      cout << '\n';
23
24          return 0;
25  }
```

La salida al ejecutar el programa

```
$ g++ -ocola -Wall -std=c++11 cola.cpp
$ ./colañ

Tamao de la cola (cola.size()): 5
Primer elemento (cola.top()): 1

Contenido :     1       3       5       7       9
```

En la linea 6 vemos que la declaración varía para indicar que de la colección guardada se vayan ordenando con la función *greater*.

En el caso de nuestro trabajo, el algoritmo de kruskal, debemos tener una cola de prioridad de aristas, de esa forma ordenaremos al grafo con las aristas de menor a mayor peso:

```
1   /**
2    * \type arbol
3    * \brief lista de prioridad de aristas ordenada de menor a
         mayor costo
4    *
5    * Arbol es una priority_queue de aristas.
6    * El criterio de ordenamiento lo da MenorValor la cual
         establece la
7    * ócomparacin de los costos de las aristas.
8    */
9   typedef priority_queue<arista, vector<arista>,MenorValor>
        arbol;
10
11  /**
12   * \class MenorValor
13   *
14   ^ \brief Clase que fija el criterio de ordenamiento a la
         cola de prioridad
15   *
16   * la priority_queue definida como tipo arbol águardar las
         aristas ordenadamente
17   * úsegn su peso de menor a mayor para luego poder ser usada
         en el algoritmo
```

```
18   *
19   */
20  class MenorValor
21  {
22  public:
23                  /**
24                   * \brief sobrecarga al operador () para la
                         ócomparacin de pesos
25                   * \param e1 de tipo arista
26                   * \param e2 de tipo arista
27                   * \return true en caso que el peso de e1 sea
                          mayor al de e2
28                   */
29          bool operator()(arista e1, arista e2) {
30              return e1.second > e2.second;
31          }
32  };
```

Vemos que se declara una clase con el sentido de generar un nuevo criterio debido a la complejidad de la estructura que se compara.