Computing Science 204 (B1)

Kruskal's Algorithm for Minimal Spanning Trees

Thursday, August 10

Given a graph $G = (V, E)$, with vertex set V and edge set E, with each edge $e = \{a, b\}$ of this graph we assign a positive real number called the *weight* of e and denoted by wt(e) or wt($\{a, b\}$). The graph G with such an assignment of weights to its edges is called a *weighted* graph (also called a *network*).

A *tree* is a graph which is connected and contains no cycles. A *spanning tree* of a graph G is a subgraph which is a tree and which spans G, that is, contains every vertex of G.

Kruskal's algorithm is a greedy algorithm which generates a minimal spanning tree of a weighted graph G, that is, a spanning subgraph of G which forms a tree and which has the property that no other spanning tree has a smaller sum of weights on its edges. The algorithm will either find a minimal spanning tree or conclude that the network does not have a spanning tree, that is, is not connected.

Given a weighted graph $G = (V, E)$, Kruskal's algorithm finds a spanning tree which minimizes the sum of the weights of the edges by building up the tree one edge at a time. It is *greedy* in the sense that it always adds edges of smallest weight first. An edge is then added to the tree if and only if it does not form a cycle with some of the other edges that have already been added. The algorithm is stated formally below.

Kruskal's Algorithm
Input: A weighted graph $G = (V, E)$.
Output: A set of edges defining a minimal spanning tree of G or the message that G is disconnected.

(i) Order the edges of G in increasing order of weight. (Order arbitrarily in case of ties.) Set $T = \emptyset$. (T is the set of edges of the minimum spanning tree.)

(ii) Add the first edge to the set T.

(iii) If every edge of G has been examined, stop and output the message that the graph G is disconnected. Otherwise, examine the first unexamined edge in the order and include it in the set T if and only if it does not form a cycle with some edges already in T. If the edge is added to T, go to (iv). Otherwise, repeat (iii).

(iv) If T has $n - 1$ edges (where n is the number of vertices of G), stop and output T. Otherwise, return to (iii).

The following *C* program implements the above algorithm, it takes as its input a weighted undirected graph G, and has as its output a minimum spanning tree T for the graph G.

The input consists of data representing a number of weighted undirected graphs, each specified by

- A natural number N, where $N \leq 100$, indicating the number of vertices in the graph (the vertices will be labeled $1, 2, 3, \cdots, N$) and `0` signals the end of data for the vertices.
- A number of triples `a b c` where the pair `a b` represents an edge as follows: The elements `a b` are natural numbers indicating the vertices which are the endpoints of the edge; and the positive real number `c` represents the weight of this edge. The end of the data for the current graph will be signalled by the triple `0 0 0`.

For each graph the output is:

- A list of the vertices and the edges with their weights (i.e. echo the input).
- If the graph is not connected, a message so indicating.
- If the graph is connected, a list of the edges in the minimal spanning tree together with the sum of the weights.

The program given below uses depth first search to detect cycles in the subgraphs generated when constructing a minimal spanning tree using Kruskal's algorithm. It is also possible to implement Kruskal's algorithm using disjoint set structures and the union–find algorithm to grow the tree.

```
/* KRUSKAL'S ALGORITHM TO FIND A MINIMUM SPANNING TREE */
   /* USES DEPTH FIRST SEARCH TO CHECK FOR CYCLES */

#include<stdio.h>
#define maxV 100
#define maxE (maxV*(maxV-1)/2)

struct edge
{
   int v1;
   int v2;
   int w;
   int in_tree;
} *edges;

struct node
{
   int v;
   struct node *next;
} *z,*adj[maxV];

int q,V,E,End,weight;
int cycle = 0, dfsn = 0;
int mark[maxV];

build_edge_list()
{
   int j,x,y,w;
   printf("Enter the number of Vertices followed by 0:\n");
   scanf("%d%d",&V,&End);
   edges = (struct edge *)malloc(maxE*sizeof(struct edge));
   printf("Enter the edges and weights: x y w\n");
   scanf("%d%d%d",&x,&y,&w);
   j = 0;
   while (x+y+w != 0)
   {
       edges[j].v1 = x;    /* build the list of edges */
       edges[j].v2 = y;
       edges[j].w = w;
       edges[j].in_tree = 0;
       j++;
       scanf("%d %d %d",&x,&y,&w);
   }
   E = j;   /* Number of Edges */
}
```

```
build_spanning_tree()
{
   int i;
   struct node *t;

   z = (struct node *)malloc(sizeof *z);
   z->next = z;
   for (i = 0; i < maxV; i++)
       adj[i] = z;
   cycle = 0; q = 0; i = 0;
   while (i < E && q < V-1)
   {   if (i >= 2)
           cycle_check(edges[i].v1,adj);
       if (cycle == 0)
       {   t = (struct node *) malloc(sizeof *t);
           t->v = edges[i].v2;
           t->next = adj[edges[i].v1];
           adj[edges[i].v1] = t;
           t = (struct node *) malloc(sizeof *t);
           t->v = edges[i].v1;
           t->next = adj[edges[i].v2];
           adj[edges[i].v2] = t;
           edges[i].in_tree = 1;
           q++;
       }
       i++;
   }
   if (q == V-1)
       print_output(edges,E);
   else
       printf("The graph is not connected. It has no spanning tree.\n");
}

sort_edge_list(L,k)
int k;
struct edge *L;
{
   int i,j,v1,v2,w;

   for (i = 1; i < k; i++)
   {
       v1 = L[i].v1;
       v2 = L[i].v2;
       w = L[i].w;
       j = i -1;
       while (j >= 0 && L[j].w > w)
       {
           L[j+1].v1 = L[j].v1;
           L[j+1].v2 = L[j].v2;
           L[j+1].w = L[j].w;
           j = j - 1;
       }
       L[j+1].v1 = v1;
       L[j+1].v2 = v2;
       L[j+1].w = w;
   }
}
```

```
cycle_check(v,B)
int v;
struct node *B[maxV];
{
   int k,parent;
   struct node *t;

   dfsn++;
   mark[v] = dfsn;
   t = B[v];
   parent = v;
   while (t != z && cycle == 0)
   {
      k = t->v;
      if (mark[k] == 0)
          cycle_check(k,B);
      else if (mark[parent] - 1 < mark[k])
          cycle = 1;
      else
          cycle = 0;
      t = t->next;
      parent = k;
   }
}

echo_input(e,N)
struct edge *e;
int N;
{
   int k;
   printf("\nThe Edges Input:\n");
   for (k = 0; k < N; k++)
   {
    printf("edge %d: %4d%4d weight: %4d\n",k+1,e[k].v1,e[k].v2,e[k].w);
   }
   printf("Number of Edges: %d\n",N);
}

print_output(e,N)
struct edge *e;
int N;
{
   int k;
   weight = 0;
   printf("\nThe Minimal Spanning Tree:\n");
   for (k = 0; k < N; k++)
   {
       if (e[k].in_tree == 1)
       {
           printf("edge %d: %4d%4d weight: %4d\n",k+1,e[k].v1,e[k].v2,e[k].w);
           weight = weight + e[k].w;
       }
   }
   printf("The Weight of a Minimal Spanning Tree is: %d\n",weight);
   printf("Number of Edges in the Minimal Spanning Tree is: %d\n",q);
}
```

```
main()
{
   build_edge_list();
   echo_input(edges,E);
   sort_edge_list(edges,E);
   build_spanning_tree();
}
```