Algorithm Lab With C
in Linux

Minimum Spanning Tree: Prim's Algo.

ANUPAM PATTANAYAK[1]
M.C.A., M. Tech.
Assistant Professor,
Department of Computer Science,
Raja N. L. Khan Women's College (Autonomous),
Midnapore, West Bengal - 721102

April 19, 2020

[1]anupam.pk@gmail.com

Contents

1

Minimum Spanning Tree

Recall that a graph G is defined by $G\,(V, E)$ where V is the set of vertices and E is the set of edges. Consider the following graph in figure 1.1. Here, $V =$

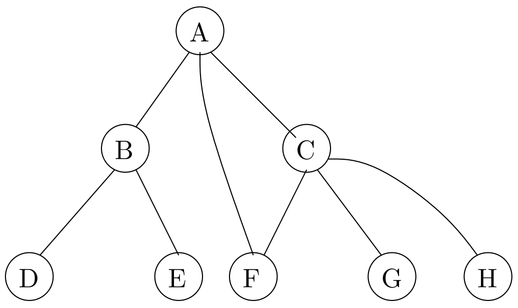

Figure 1.1: A Graph

$\{A, B, C, D, E, F, G, H\}$, and $E = \{AB, AC, AF, BD, BE, CF, CG, CH\}$. Here the edges are bidirectional. Two vertices X and Y are said to be *adjacent* if there is an edge between vertices X and Y.

1.1 Spanning Tree

Spanning tree is a subset of graph G that spans over the entire vertex set of the G. Following program finds the spanning tree of an undirected graph where each edge has unit cost.

```
/* C prog for Spanning Tree of an undirected unweighted graph */
 /* cycle detection is done by inspecting adjacency matrix */

#include <stdio.h>
```

```
#include <stdlib.h>

int main() {
 int n,i,j;
 int **adj;   /* adjacency matrix */
 int **adj_copy;
 int **adj_sp; /* adjacency matrix of SP */
 int *vertex_span;  /* keeps track of vertex inclusion in SP */
 int sp_edge_count;  /* keeps track of number of edges in
   growing SP */

 printf("\n Enter Number of Vertices in Graph: ");
 scanf("%d",&n);

 adj=(int**)malloc(n*sizeof(int*));  /* allocate memory for
   adjacent matrix */
 adj_copy=(int**)malloc(n*sizeof(int*));
 adj_sp=(int**)malloc(n*sizeof(int*));
 vertex_span=(int*)malloc(n*sizeof(int));

 for(i=0;i<n;i++) {
   adj[i]=(int*)malloc(n*sizeof(int));
   adj_copy[i]=(int*)malloc(n*sizeof(int));
   adj_sp[i]=(int*)malloc(n*sizeof(int));

 }

 printf("\n Enter Adjacency Matrix of the Graph: ");
 for(i=0;i<n;i++) {
  for(j=0;j<n;j++)  {
   scanf("%d",&adj[i][j]);
   adj_copy[i][j]=adj[i][j];
   adj_sp[i][j]=0;
  }
 }

 printf("\nn=%d ",n);
 printf("\n Adjacency Matrix: ");
 for(i=0;i<n;i++)  {
  printf("\n");
  for(j=0;j<n;j++)
   printf("%5d",adj[i][j]);
 }

 for(i=0;i<n;i++)    /* lower triangular all 0, if edge A-B
   present then ignore B-A */
  for(j=0;j<i;j++)
    adj_copy[i][j]=0;
```

```
printf("\n Adjacency Matrix, with lower triangular 0: ");
for(i=0;i<n;i++)  {
 printf("\n");
 for(j=0;j<n;j++)
  printf("%5d",adj_copy[i][j]);
}

for(i=0;i<n;i++)    /* initialize vertex_span[] */
   vertex_span[i]=0;

              /* create ST */
for(i=0;i<n;i++)  {
 if(i==0) {
   vertex_span[i]=1;
   sp_edge_count=1;
   for(j=0;j<n;j++) {
       if(adj_copy[i][j]==1) {
          adj_sp[i][j]=1;
          vertex_span[j]=1;
          sp_edge_count++;
       }
   }
   if(sp_edge_count==(n-1))
     break;
 }
 else {
    for(j=i+1;j<n;j++) {
      if((adj_copy[i][j]==1)&&(vertex_span[j]==0)) {
          adj_sp[i][j]=1;
          vertex_span[j]=1;
          sp_edge_count++;
       }
       if(sp_edge_count==(n))
         break;
    }
 }

}

printf("\n Spanning Tree Adjacency Matrix: ");
for(i=0;i<n;i++)  {
 printf("\n");
 for(j=0;j<n;j++)
  printf("%5d",adj_sp[i][j]);
}

printf("\n");

return 0;
```

```
}
```

We will redirect the input from a text file. We store graph in terms of adjacency matrix in a text file named *graph_ip1.txt* as follows.

```
8
0 1 1 0 0 1 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 1 1
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0
```

First input is the number of vertices in input graph. Next is the 8×8 adjacency matrix of graph of figure 1.1. Last number is the source vertex from where graph traversal is to begin.

Now, we compile and execute the program. A sample output corresponding to the graph shown in figure 1.1 is given next.

```
$ gcc msp_prim.c
$ ./a.out<graph_ip1.txt

 Enter Number of Vertices in Graph:
 Enter Adjacency Matrix of the Graph:
n=8
 Adjacency Matrix:
    0    1    1    0    0    1    0    0
    1    0    0    1    1    0    0    0
    1    0    0    0    0    1    1    1
    0    1    0    0    0    0    0    0
    0    1    0    0    0    0    0    0
    1    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
 Adjacency Matrix, with lower traingular 0:
    0    1    1    0    0    1    0    0
    0    0    0    1    1    0    0    0
    0    0    0    0    0    1    1    1
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
```

```
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
 Spanning Tree Adjacency Matrix:
      0      1      1      0      0      1      0      0
      0      0      0      1      1      0      0      0
      0      0      0      0      0      0      1      1
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
$
```

The spanning tree obtained can be visualized and checked as of following figure 1.2, shown in color.

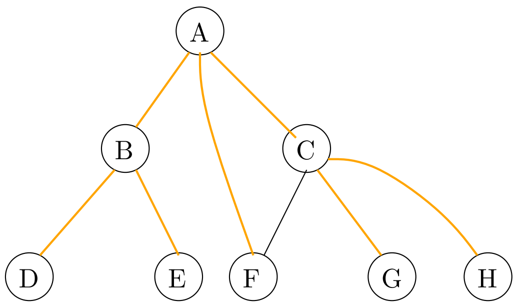

Figure 1.2: A Spanning Tree

1.2 Prim's Algorithm

Prim's algorithm is a minimum spanning tree (MST) finding algorithm. It starts from a vertex, and continue to grow the tree by taking edge with minimum weight adjacent to any node in the growing tree. We have to be careful so that no circuit gets formed while expanding the tree. When we finish adding n-1 edges, we obtain a minimum spannig tree.

The input weighted graph considered is given in figure ??. In the following, we give the C program for Prim's algorithm.

```
/* C prog for Minimum Spanning Tree of an undirected graph by
   Prim's Algorithm */
```

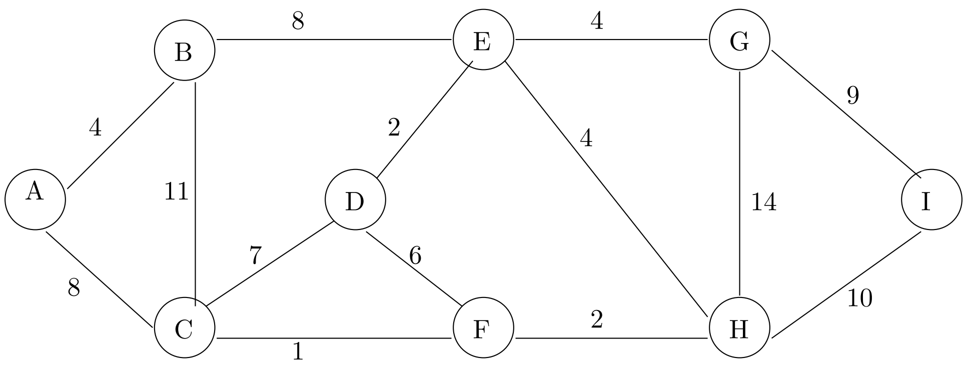

Figure 1.3: An Weighted Graph

```
 /* cycle detection is done inspecting adjacency matrix */

#include <stdio.h>
#include <stdlib.h>

struct mat_index{
 int i;
 int j;
};

int main() {
 int n,i,j,k;
 int **adj;   /* adjacency matrix */
 int **adj_copy;
 int **adj_msp; /* adjacency matrix of MSP */
 int *vertex_span;  /* keeps track of vertex inclusion in MSP */
 int msp_edge_count=0;  /* keeps track of number of edges in
   growing MSP */

 int vertex_count(int*, int);   /* counts number of vertices
   included so far */
 struct mat_index min_adj_matrix(int**, int*, int);
 struct mat_index loc;

 printf("\n Enter Number of Vertices in Graph: ");
 scanf("%d",&n);

 adj=(int**)malloc(n*sizeof(int*));  /* allocate memory for
   adjacent matrix */
 adj_copy=(int**)malloc(n*sizeof(int*));
 adj_msp=(int**)malloc(n*sizeof(int*));
 vertex_span=(int*)malloc(n*sizeof(int));
```

```
for(i=0;i<n;i++) {
  adj[i]=(int*)malloc(n*sizeof(int));
  adj_copy[i]=(int*)malloc(n*sizeof(int));
  adj_msp[i]=(int*)malloc(n*sizeof(int));

}

printf("\n Enter Adjacency Matrix of the Graph: ");
for(i=0;i<n;i++) {
 for(j=0;j<n;j++)  {
  scanf("%d",&adj[i][j]);
  adj_copy[i][j]=adj[i][j];
  adj_msp[i][j]=100;
 }
}

printf("\nn=%d ",n);
printf("\n Adjacency Matrix: ");
for(i=0;i<n;i++)  {
 printf("\n");
 for(j=0;j<n;j++)
  printf("%5d",adj[i][j]);
}

for(i=0;i<n;i++)    /* lower traingular all 0, if edge A-B
   present then ignore B-A */
 for(j=0;j<i;j++)
   adj_copy[i][j]=100;

printf("\n Adjacency Matrix, with lower traingular initialized
   with high value 100: ");
for(i=0;i<n;i++)  {
 printf("\n");
 for(j=0;j<n;j++)
  printf("%6d",adj_copy[i][j]);
}

for(i=0;i<n;i++)    /* initialize vertex_span[] */
   vertex_span[i]=0;

                /* create MSP */
vertex_span[0]=1;
for(i=0;i<n;i++)  {
   loc=min_adj_matrix(adj_copy, vertex_span, n);
   adj_msp[loc.i][loc.j]=adj_copy[loc.i][loc.j];
   adj_copy[loc.i][loc.j]=100;   /* initialize with high value,
    so that it is no more considered*/
   vertex_span[loc.j]=1;
   msp_edge_count++;
```

```
    printf("\n i=%d: vertex_span[]= ",i);
    for(k=0;k<n;k++)
      printf("%d ",vertex_span[k]);
    printf("adj_msp[%d][%d]=%d \n",loc.i,loc.j,adj_msp[loc.i][
   loc.j]);
    if(msp_edge_count==(n-1))
      break;
 }

 printf("\n MSP Adjacency Matrix: ");
 for(i=0;i<n;i++)  {
  printf("\n");
  for(j=0;j<n;j++)
   printf("%5d",adj_msp[i][j]);
 }

 printf("\n");

 return 0;
}

struct mat_index min_adj_matrix(int** B, int* rows, int n) {
   /* Function Definition */
 int i,j,mn;
 struct mat_index index;

 mn=100; /* initialize first element is the minimum */
 for(i=0;i<n;i++) {
   if(rows[i]==1)  {
    for(j=0;j<n;j++)  {
      if((B[i][j] <= mn)&& (detect_cycle(B,rows,n,i,j)==0)) {
         mn=B[i][j];
         index.i=i;
         index.j=j;
      }
     }
    }
  }
 return index;
}

int vertex_count(int* v, int n) {   /* counts number of vertices
    included in MST so far */
 int i,sum=0;
 for(i=0;i<n;i++)
    if(v[i]==1)
      sum++;
 return n;
}
```

```
int detect_cycle(int** adj_mat, int* v, int n, int a, int b) {
    /* does inclusion of a-b make a cycle? */
  int i;
  for(i=0;i<n;i++)
    if(v[b]==1)
      return 1;
  return 0;
}
```

Hopefully, you have typed the program correctly and now it is the time for compilation and execution. Here also, we will redirect the input from a text file. We store the adjacency matrix of the graph shown in figure **??** in a text file named *mst_graph_ip1.txt* as follows.

```
9
100 4 100 100 100 100 100 8 100
4 100 8 100 100 100 100 11  100
100 8 100 7 100 4 100 100 2
100 100 7 100 9 14  100 100 100
100 100 100 9 100 10  100 100 100
100 100 4 14  10  100 2 100 100
100 100 100 100 100 2 100 1 6
8 11  100 100 100 100 100 100 7
100 100 2 100 100 100 6 7 100
```

First input is the number of vertices in input graph. Next is the 9 × 9 adjacency matrix of graph of figure 1.1. The output is shown next.

Now, we compile and execute the program. A sample output corresponding to the graph shown in figure 1.1 is given next.

```
$ gcc msp_prim1.c
$ ./a.out<mst_graph_ip1.txt

 Enter Number of Vertices in Graph:
 Enter Adjacency Matrix of the Graph:
n=9
 Adjacency Matrix:
  100     4   100   100   100   100   100     8   100
    4   100     8   100   100   100   100    11   100
  100    8  100    7  100    4  100  100    2
  100  100    7  100    9   14  100  100  100
  100  100  100    9  100   10  100  100  100
  100  100    4   14   10  100    2  100  100
  100  100  100  100  100    2  100    1    6
    8   11  100  100  100  100  100  100    7
  100  100    2  100  100  100    6    7  100
 Adjacency Matrix, with lower triangular initialized with high
    value 100:
    100     4   100   100   100   100   100     8   100
    100   100     8   100   100   100   100    11   100
    100   100   100     7   100     4   100   100     2
    100   100   100   100     9    14   100   100   100
    100   100   100   100   100    10   100   100   100
    100   100   100   100   100   100     2   100   100
    100   100   100   100   100   100   100     1     6
    100   100   100   100   100   100   100   100     7
    100   100   100   100   100   100   100   100   100
 i=0: vertex_span[]= 1 1 0 0 0 0 0 0 0 adj_msp[0][1]=4

 i=1: vertex_span[]= 1 1 1 0 0 0 0 0 0 adj_msp[1][2]=8

 i=2: vertex_span[]= 1 1 1 0 0 0 0 0 1 adj_msp[2][8]=2

 i=3: vertex_span[]= 1 1 1 0 0 1 0 0 1 adj_msp[2][5]=4

 i=4: vertex_span[]= 1 1 1 0 0 1 1 0 1 adj_msp[5][6]=2

 i=5: vertex_span[]= 1 1 1 0 0 1 1 1 1 adj_msp[6][7]=1

 i=6: vertex_span[]= 1 1 1 1 0 1 1 1 1 adj_msp[2][3]=7

 i=7: vertex_span[]= 1 1 1 1 1 1 1 1 1 adj_msp[3][4]=9

 MSP Adjacency Matrix:
  100    4  100  100  100  100  100  100  100
  100  100    8  100  100  100  100  100  100
  100  100  100    7  100    4  100  100    2
  100  100  100  100    9  100  100  100  100
  100  100  100  100  100  100  100  100  100
  100  100  100  100  100  100    2  100  100
  100  100  100  100  100  100  100    1  100
  100  100  100  100  100  100  100  100  100
  100  100  100  100  100  100  100  100  100
$
```

The MST obtained can be visualized and checked as of following figure 1.4, shown in color.

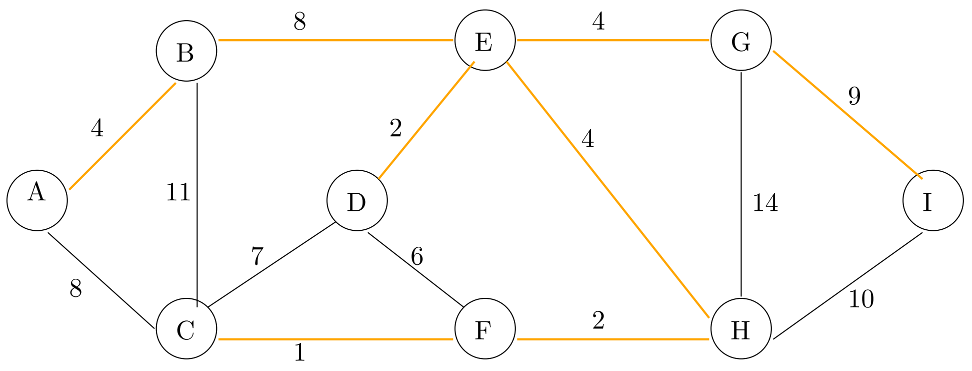

Figure 1.4: A Minimum Spanning Tree obtained by Prim's Algorithm

Create the adjacency matrix of any other graph and test the program execution on it.