CS29003 ALGORITHMS LABORATORY

Minimum Spanning Tree

Last Date of Submission: 1 – November – 2018

General Instruction

1. Please do not use any global variable unless you are explicitly instructed so.
2. Please use proper indentation in your code.
3. Please name your file as `<roll_no>_<assignment_no>`. For example, if your roll number is 14CS10001 and you are submitting assignment 3, then name your file as `14CS10001_3.c or 14CS10001_3.cpp` as applicable.
4. Please write your name and roll number at the beginning of your program.

We have already learnt Kruskal's and Prim's algorithms for finding a minimum spanning tree (MST hereafter) of a weighted undirected graph. In this exercise, we will learn another interesting algorithm for finding an MST of a weighted undirected graph. The algorithm works as follows. Assume that the input graph is connected. Initially all the edges of the input graph are unmarked. We pick an unmarked edge f of highest weight in the graph. If deleting f from the current graph makes it disconnected, then mark f; otherwise delete f from the current graph. When the graph does not contain any unmarked edge, output the current graph as an MST of the input graph (can you see why the algorithm always outputs a tree?). Figure 1 shows a run of the algorithm on a graph.

Part I. Read the graph from the user

In this part, you will read the number n of nodes and m of edges from the user. After reading n from the user, dynamically create a $n \times n$ 2-D array $\mathcal{A}$. The nodes are then numbered as $0, 1, \cdots, n-1$ and you will read the list of neighbors for each of the n nodes along with the edge weights in that order. Store the neighborhood information in the adjacency matrix $\mathcal{A}[n][n]$ format. That is, if there is an edge between nodes i and j of weight w, then $\mathcal{A}[i][j] = \mathcal{A}[j][i] = w$; other entries are 0. Print the neighbours of every node in the graph. Define the following typedef. Assume that the input graph is undirected, connected, and the weight of every edge is some positive integer.

```
typedef int **graph;
/* A variable of type graph will point to the adjacency matrix
of the graph which you will create dynamically
after reading the number of nodes from the user. */
```

Part II: A Graph Connectivity Checking Algorithm Using BFS

In this part, you use Breadth First Search (BFS) from any node of the graph to find out whether the graph is connected. Please use the following prototype. It returns 1 if the input graph is connected and 0 otherwise. Your algorithm should run in time $\mathcal{O}(m+n)$.

int isConnectedUsingBFS(graph G, int numberOfNodes)

Part III: Implement the MST Finding Algorithm

In this part, use the function you have written in Part II to implement the MST finding algorithm discussed above. Your algorithm should run in time $\mathcal{O}(m(m+n))$.

void findMST(graph G, int numberOfNodes)

The function takes a graph as input and prints the MST in G.

main()

1. Read n and m from the user. Create a 2-D array G of size $n \times n$.
2. Read the number and list of neighbors for each of the nodes and create the adjacency matrix representation of the graph G. Then print the neighbours of every node in the graph G. You can assume that the input corresponds to some undirected graph (properties like sum of degrees should be even, etc. hold); you do not need to check it. The input format is as shown below. See sample output for concreteness.

   ```
   Neighbors of i: <neighbor 1> <weight of edge to neighbor 1> <neighbor 2>
   <weight of edge to neighbor 2> ...
   ```

3. Call the function findMST(.,.) and print the the neighbours of every node in an MST of the graph G.

Sample Output 1:

The graph in the sample output is the one depicted in Figure 1. Also the sample output is the MST shown in Figure 1.

```
Enter the number of nodes and edges
7 11
Enter the neighbors of each node and corresponding weights
Degree of Vertex 0: 2
Neighbors of 0: 3 5 1 7
Degree of Vertex 1: 4
Neighbors of 1: 0 7 3 9 2 8 4 7
Degree of Vertex 2: 2
Neighbors of 2: 1 8 4 5
Degree of Vertex 3: 4
Neighbors of 3: 0 5 1 9 4 15 5 6
Degree of Vertex 4: 5
Neighbors of 4: 1 7 2 5 3 15 5 8 6 9
Degree of Vertex 5: 3
Neighbors of 5: 3 6 4 8 6 11
Degree of Vertex 6: 2
Neighbors of 6: 5 11 4 9
The neighbors of each node in the input graph and corresponding weights
Vertex 0: 3 5 1 7
```

```
Vertex 1: 0 7 3 9 2 8 4 7
Vertex 2: 1 8 4 5
Vertex 3: 0 5 1 9 4 15 5 6
Vertex 4: 1 7 2 5 3 15 5 8 6 9
Vertex 5: 3 6 4 8 6 11
Vertex 6: 5 11 4 9
The neighbors of each node of the MST you compute and corresponding weights
Vertex 0: 1 7 3 5
Vertex 1: 0 7 4 7
Vertex 2: 4 5
Vertex 3: 0 5 5 6
Vertex 4: 1 7 2 5 6 9
Vertex 5: 3 6
Vertex 6: 4 9
```

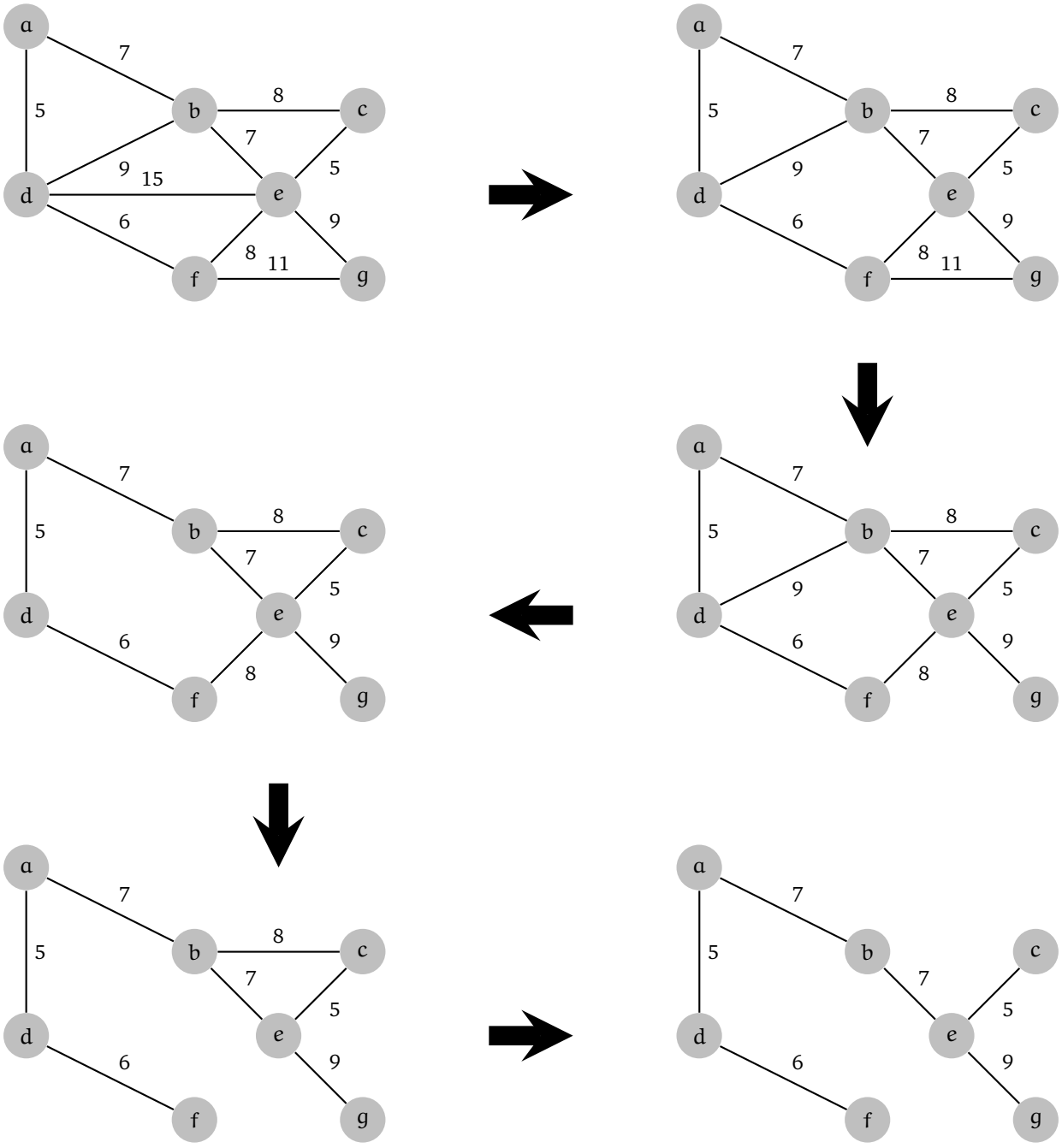

Figure 1: Example of a run of the algorithm.