Kruskal's MST Algorithm

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm

Description

- create a forest *T* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and T is not yet spanning
    - remove an edge with minimum weight from *S*
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

The above algorithm outline can be more precisely described with:

Alternate Description

```
KRUSKAL(G):
1 T = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6       T = T ∪ {(u, v)}
7       UNION(u, v)
8 return T
```

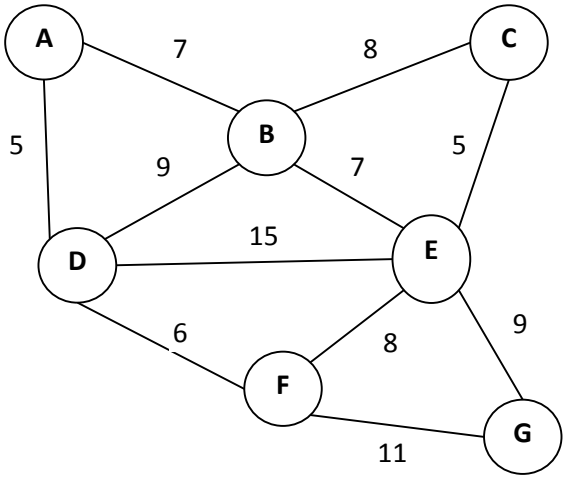
Data Structure Consideration

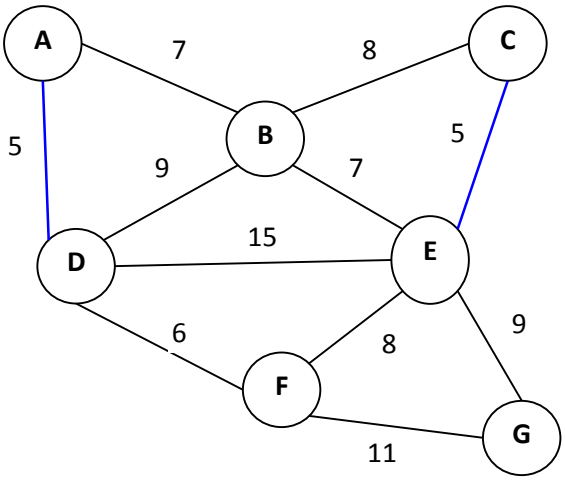
To implement this algorithm a way to represent sets, to find which set a vertex is in and to get the union of two sets is required. We refer to a data structure which supports this as a Union-Find data strucure. For junior undergraduates, it may seem difficul to implement. Two standard implementation are available: Disjoint-set linked lists and Disjoint-set trees.

Example 1

wgraph3.txt

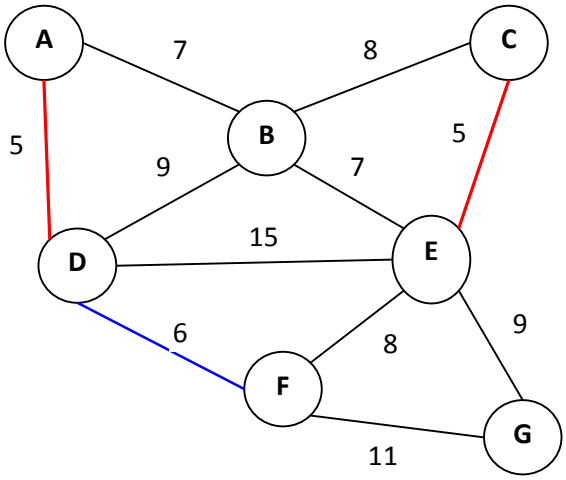
Initially all 11 edges are in the heap and each vertex is isolated. Then after next 2 minimum edges A-5-D and C-5-E are chosen as shown.



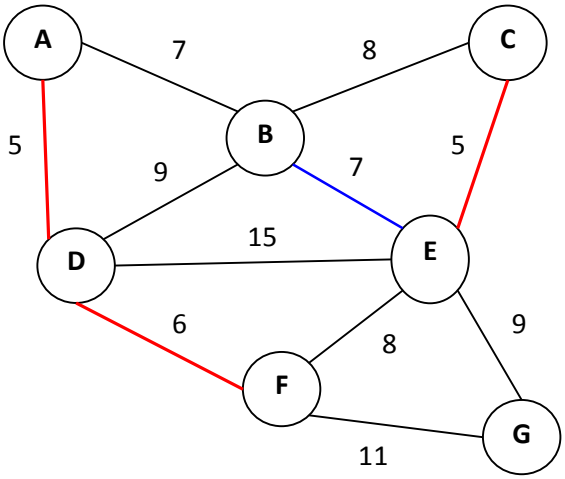


MST trees are { {A} {B} {C} {D} {E} {F} {G}}.

Then D-6-F and B-7-E chosen.

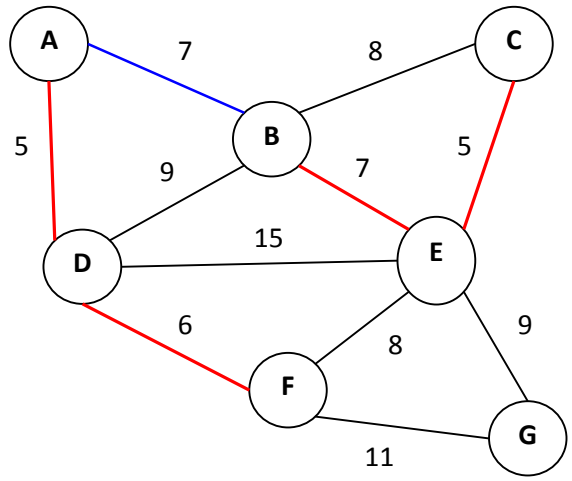


MST trees { {A D} {B} {C E} {F} {G}}

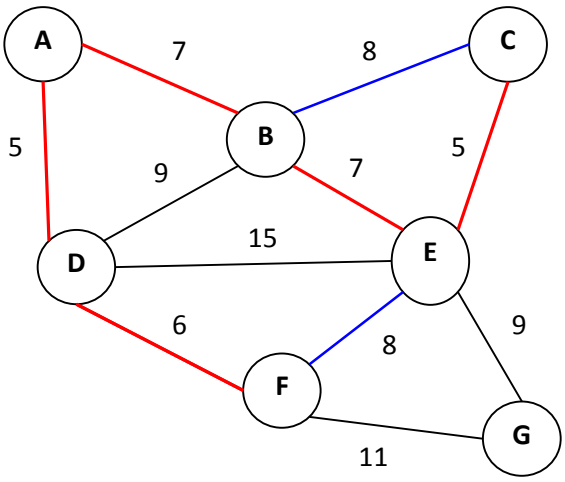


MST trees { {A D F} {B} {C E} {G}}

Next A-7-B is chosen which joins 2 trees.

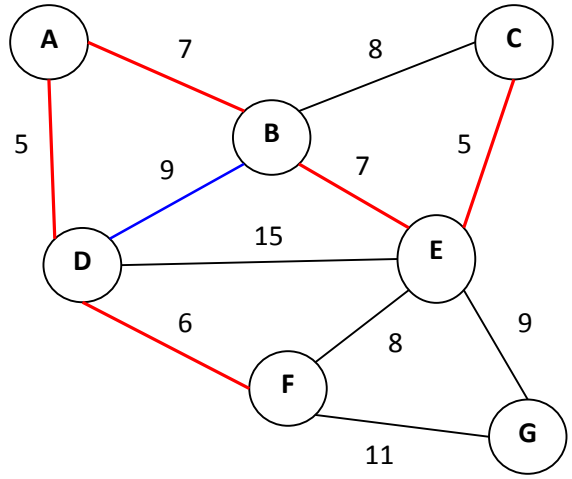


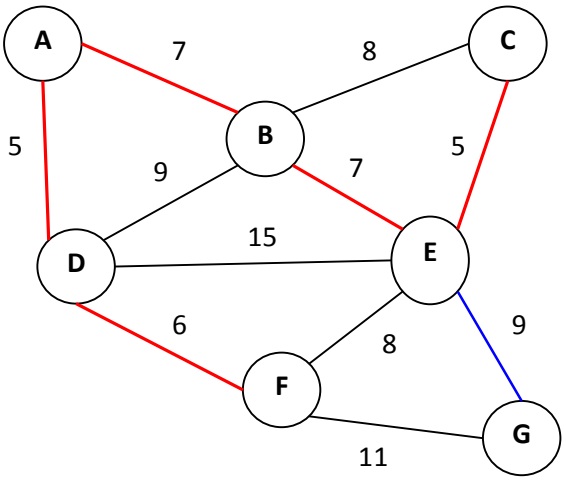
MST trees { {A D F} {C E B} {G}}



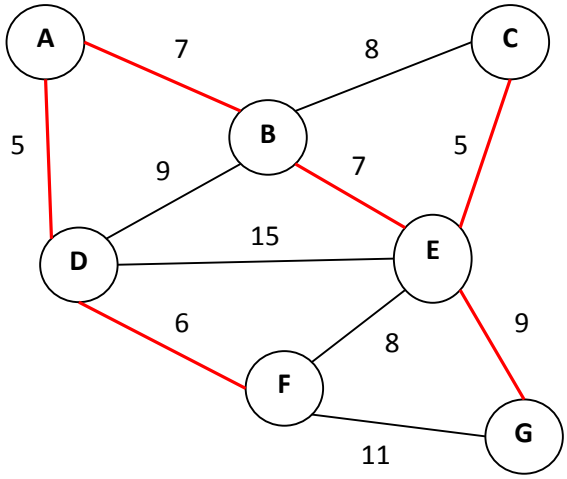
MST trees { {A D F C E B} {G}}.

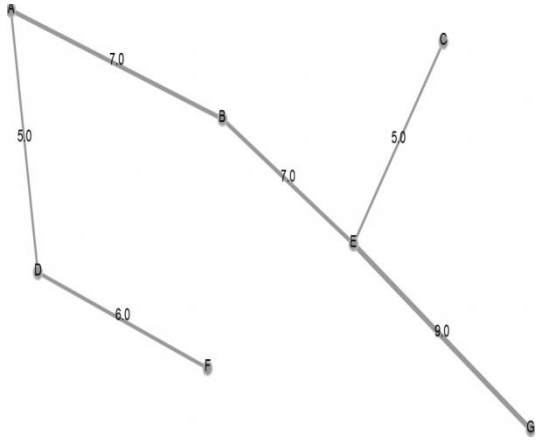
But next 3 edges chosen are B-8-C, E-8-F and D-9-B which would give rise to cycles, and so are ignored.





Edge E-9-G completes the MST, the algorithm stops here as all vertices are now in the MST. Note there are still 2 edges left, F-11-G and D-15-E, but they don't matter.

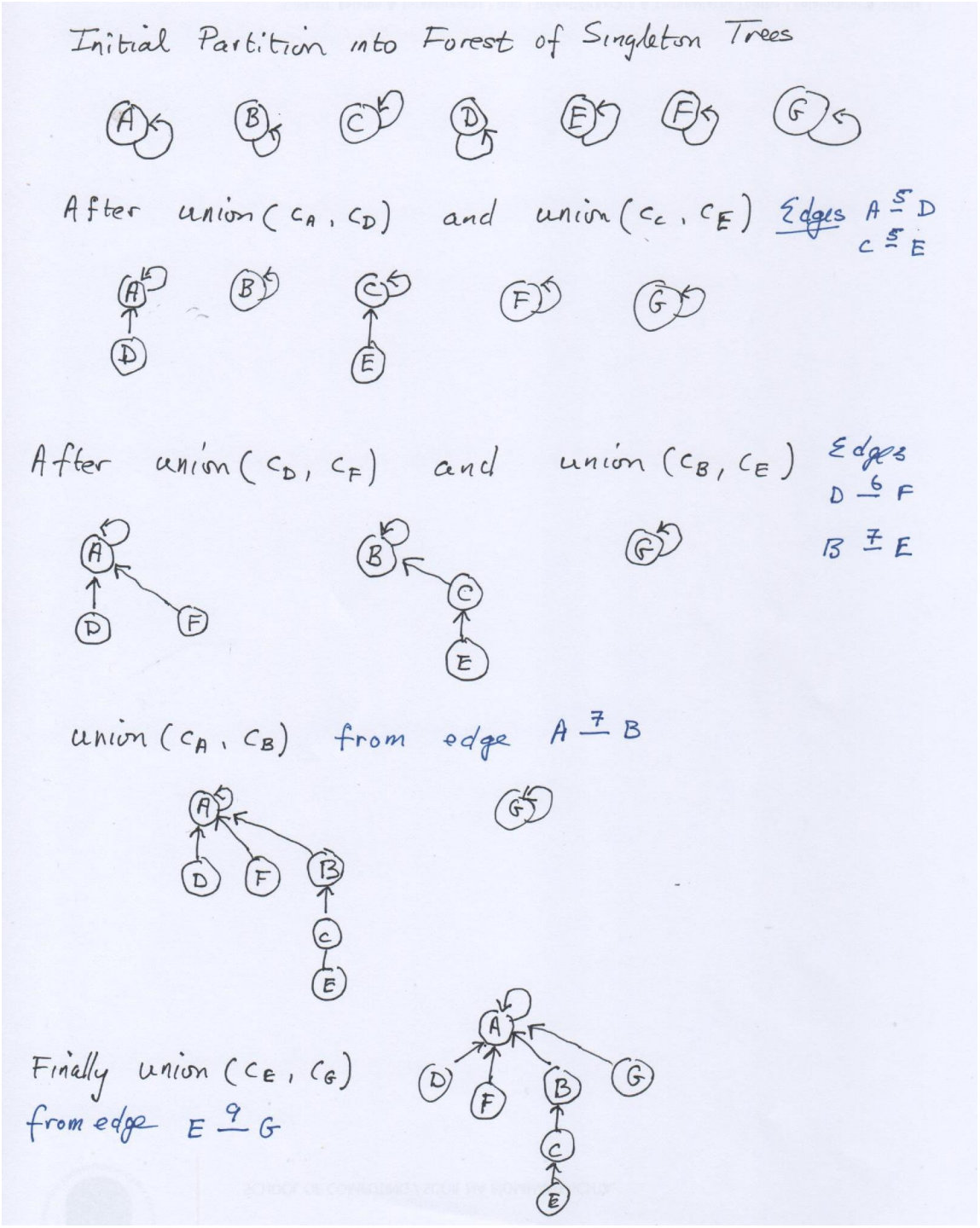




Union-find Trees Representation

When trees are used to represent the disjoint union-find sets each set is represented by a tree and the tree root is used to label the set. All the vertices on a tree are considered to be in the same set. Initially each vertex is in a singleton set and is the root of its own tree. In general a vertex points to its parent on the tree and the root vertex points to itself. This allow a traversal from a vertex to the tree root in a simple manner. A simple array parent[V+1] is sufficient for this where parent[u] indicates the vertex that u is attached to in the tree. Note parent[root] == root.

In this approach, find(u) means travelling back the tree from vertex u until the root is found and so may take more than 1 step. Once the two roots Cu and Cv are found, the union(Cu , Cv) takes 1 step and just involves linking one root to the other. These operations are illustrated below.



Algorithm in Pseudocode

```
MST_Kruskal()
begin
    // Input is simple connected graph represented by array of edges edge[]
    // Output is list of edges T in MST
    // Create a partition for the set of vertices
    foreach vertex v ∈ V
        Cv := {v}

    // create a minHeap h from array of edges E
    h := new Heap( E)

    // let T be an initially empty tree
    T := ∅
    while size(T) < n-1
        (u, v, wgt) :=  h.removeMin()
        Cv  :=  findSet(v)
        Cu  :=  findSet(u)
        if  Cu ≠ Cv
            union(Cu , Cv)
            T  :=  T ∪ {(u, v, wgt)}
    return T
end
```

Time Complexity

Kruskal's algorithm can be shown to run in $O(E \log V)$ time.
By way of comparison Prim complexity is $O(E \log V)$ for sparse and $O(V^2) = O(E)$ for dense.

So for sparse graphs the performance is similar and for dense graphs Prim on an adjacency matrix is better.

Implementation Concerns

A union-find data structure is required for Kruskal's algorithm. At any stage of the algorithm, the graph vertices are partitioned into disjoint sets. When an edge is being considered for the MST, we must first find which sets the edge vertices belong to. Hence a *findSet()* operation is required. If the vertex sets are disjoint, the edge is added to the MST and the union of the two sets is obtained. So a *union()* operation is also needed.

findSet : Vertex $\rightarrow$ Set
union : Set×Set $\rightarrow$ Set

Two effective ways to implement the union-find data structure are:

- Disjoint-set linked lists
- Disjoint-set trees.

The disjoint-set trees method when improved with *union by* ***rank*** and ***path compression*** offers the most efficient approach.

Example 2 - shows more algorithm detail

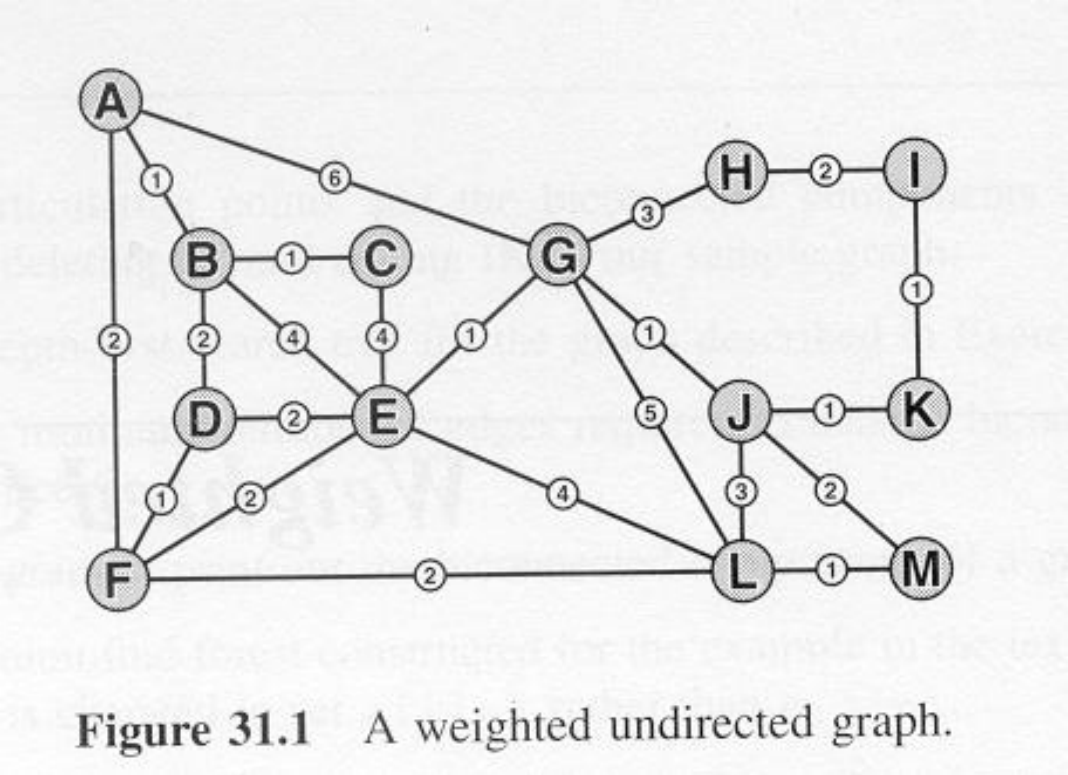
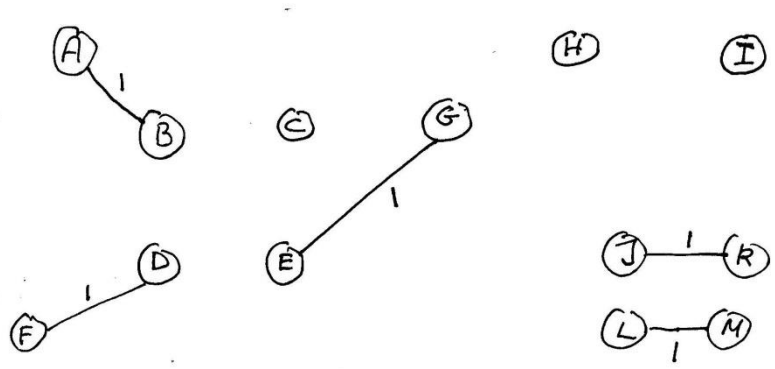


Figure 31.1 A weighted undirected graph.

Initially connected components are just singleton sets with individual vertices:
{A} {B} {C} {D} {E} {F} {G} {H} {I} {J} {K} {L} {M}

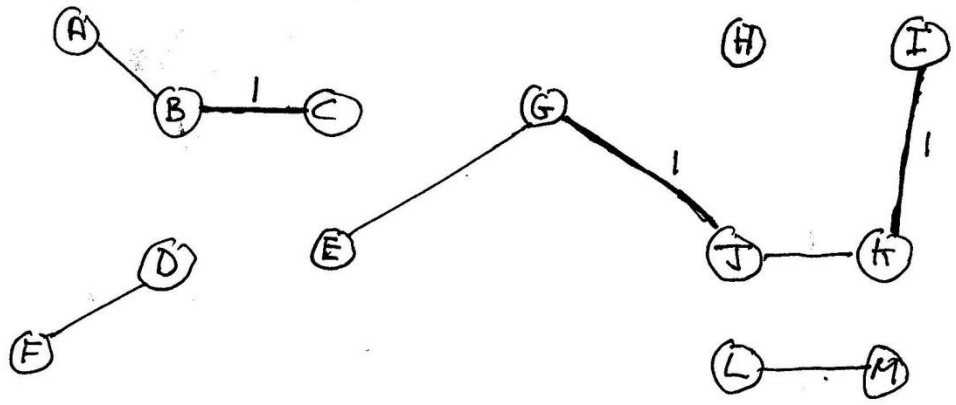
After the following five minimum edges
A-1-B
D-1-F
E-1-G
J-1-K,
L-1-M
have been added to the MST, components are:

{A,B} {D,F} {C} {E,G} {H} {I} {J,K} {L,M}



Choosing 3 more minimum edges of weight 1,
B-1-C
G-1-J
K-1-I
we will have five connected components:

{A,B,C} {D,F} {E,G,J,K,I} {H} {L,M}



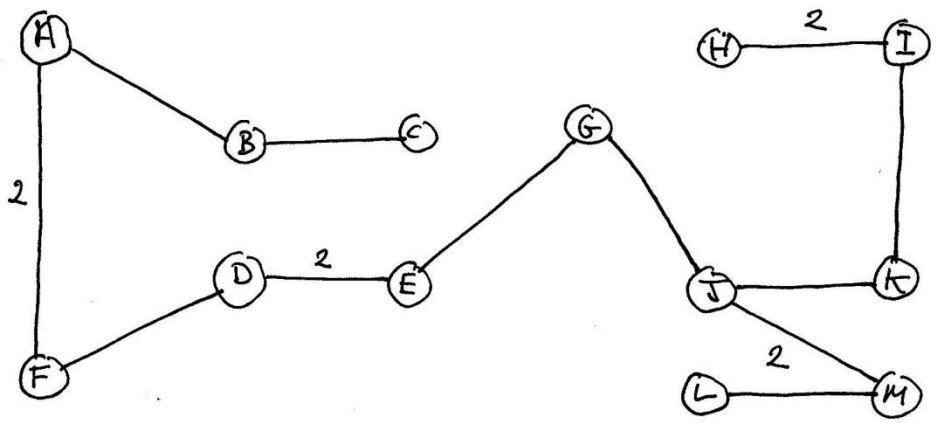
Next supposing the following minimum edges are chosen:
A-2-F
{A,B,C,D,F} {E,G,J,K,I} {H} {L,M}

H-2-I
{A,B,C,D,F} {E,G,J,K,I,H} {L,M}

J-2-M
{A,B,C,D,F} {E,G,J,K,I,H,L,M}

And finally add edge D-2-E to get one connected component::

{A,B,C,D,F,E,G,J,K,I,H,L,M}


A
B
C
D
E
F
G
H
I
J
K
L
M
2
2
2
2