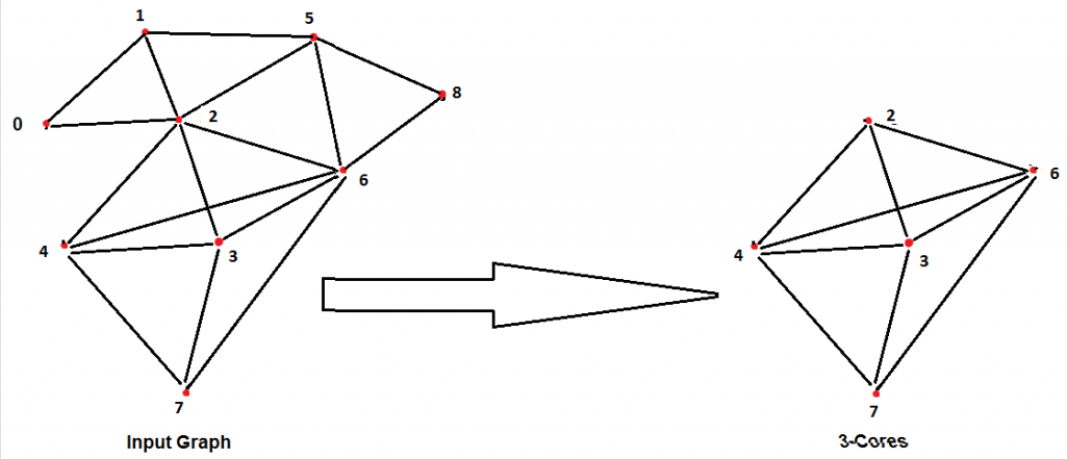
Find k-cores of an undirected graph

Given a graph G and an integer K, K-cores of the graph are connected components that are left after all vertices of degree less than k have been removed (Source wiki)

Example:

```
Input : Adjacency list representation of graph shown
        on left side of below diagram
Output: K-Cores :
[2] -> 3 -> 4 -> 6
[3] -> 2 -> 4 -> 6 -> 7
[4] -> 2 -> 3 -> 6 -> 7
[6] -> 2 -> 3 -> 4 -> 7
[7] -> 3 -> 4 -> 6
```



We strongly recommend you to minimize your browser and try this yourself first.

The standard algorithm to find a k-core graph is to remove all the vertices that have degree less than- 'K' from the input graph. We must be careful that removing a vertex reduces the degree of all the vertices adjacent to it, hence the degree of adjacent vertices can also drop below-'K'. And thus, we may have to remove those vertices also. This process may/may not go until there are no vertices left in the graph.

To implement above algorithm, we do a modified DFS on the input graph and delete all the vertices having degree less than 'K', then update degrees of all the adjacent vertices, and if their degree falls below 'K' we will delete them too.

Below is C++ implementation of above idea. Note that the below program only prints vertices of k cores, but it can be easily extended to print the complete k cores as we have modified adjacency list.

```
// C++ program to find K-Cores of a graph
#include<bits/stdc++.h>
using namespace std;

// This class represents a undirected graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);
```

```
    // A recursive function to print DFS starting from v
    bool DFSUtil(int, vector<bool> &, vector<int> &, int k);

    // prints k-Cores of given graph
    void printKCores(int k);
};

// A recursive function to print DFS starting from v.
// It returns true of degree of v after processing is less
// than k else false
// It also updates degree of adjacent if degree of v
// is less than k.  And if degree of a processed adjacent
// becomes less than k, then it reduces of degree of v also,
bool Graph::DFSUtil(int v, vector<bool> &visited,
                    vector<int> &vDegree, int k)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // degree of v is less than k, then degree of adjacent
        // must be reduced
        if (vDegree[v] < k)
            vDegree[*i]--;

        // If adjacent is not processed, process it
        if (!visited[*i])
        {
            // If degree of adjacent after processing becomes
            // less than k, then reduce degree of v also.
            if (DFSUtil(*i, visited, vDegree, k))
                vDegree[v]--;
        }
    }

    // Return true if degree of v is less than k
    return (vDegree[v] < k);
}

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Prints k cores of an undirected graph
void Graph::printKCores(int k)
{
    // INITIALIZATION
    // Mark all the vertices as not visited and not
    // processed.
    vector<bool> visited(V, false);
    vector<bool> processed(V, false);

    int mindeg = INT_MAX;
    int startvertex;

    // Store degrees of all vertices
    vector<int> vDegree(V);
    for (int i=0; i<V; i++)
    {
        vDegree[i]  = adj[i].size();
```

```
        if (vDegree[i] < mindeg)
        {
            mindeg = vDegree[i];
            startvertex=i;
        }
    }

    DFSUtil(startvertex, visited, vDegree, k);

    // DFS traversal to update degrees of all
    // vertices.
    for (int i=0; i<V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited, vDegree, k);

    // PRINTING K CORES
    cout << "K-Cores : \n";
    for (int v=0; v<V; v++)
    {
        // Only considering those vertices which have degree
        // >= K after BFS
        if (vDegree[v] >= k)
        {
            cout << "\n[" << v << "]";

            // Traverse adjacency list of v and print only
            // those adjacent which have vDegree >= k after
            // BFS.
            list<int>::iterator itr;
            for (itr = adj[v].begin(); itr != adj[v].end(); ++itr)
                if (vDegree[*itr] >= k)
                    cout << " -> " << *itr;
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    int k = 3;
    Graph g1(9);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 5);
    g1.addEdge(2, 3);
    g1.addEdge(2, 4);
    g1.addEdge(2, 5);
    g1.addEdge(2, 6);
    g1.addEdge(3, 4);
    g1.addEdge(3, 6);
    g1.addEdge(3, 7);
    g1.addEdge(4, 6);
    g1.addEdge(4, 7);
    g1.addEdge(5, 6);
    g1.addEdge(5, 8);
    g1.addEdge(6, 7);
    g1.addEdge(6, 8);
    g1.printKCores(k);

    cout << endl << endl;

    Graph g2(13);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(0, 3);
    g2.addEdge(1, 4);
    g2.addEdge(1, 5);
    g2.addEdge(1, 6);
    g2.addEdge(2, 7);
    g2.addEdge(2, 8);
```

```
    g2.addEdge(2, 9);
    g2.addEdge(3, 10);
    g2.addEdge(3, 11);
    g2.addEdge(3, 12);
    g2.printKCores(k);

    return 0;
}
```

Output :

```
K-Cores :

[2] -> 3 -> 4 -> 6
[3] -> 2 -> 4 -> 6 -> 7
[4] -> 2 -> 3 -> 6 -> 7
[6] -> 2 -> 3 -> 4 -> 7
[7] -> 3 -> 4 -> 6

K-Cores :
```

Time complexity of the above solution is O(V + E) where V is number of vertices and E is number of edges.

Related Concepts :
Degeneracy : Degeneracy of a graph is the largest value k such that the graph has a k-core. For example, the above shown graph has a 3-Cores and doesn't have 4 or higher cores. Therefore, above graph is 3-degenerate.
Degeneracy of a graph is used to measure how sparse graph is.

Reference :
https://en.wikipedia.org/wiki/Degeneracy_%28graph_theory%29