```java
//Author: Ma Hongqiang
//Email: mhq199657@163.com
import java.util.ArrayList;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Queue;
import java.util.LinkedList;
import java.util.Stack;
import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Comparator;
import java.util.Arrays;
class Graph<E extends Comparable<E>>{
  public boolean isDirected;
  public static final int NO_VERTEX = 0;
  public static final int UNWEIGHTED = 1;
  public static int UID = 0;
  public static final int INF = Integer.MAX_VALUE;
  public ArrayList<ArrayList<Integer>> _adjacencyMatrix;
  public ArrayList<ArrayList<IntegerPair>> _adjacencyList;
  public ArrayList<IntegerTriple> _edgeList;
  public ArrayList<Vertex<E>> _vertexList;
  HashMap<E, Integer> _vertexMap;
  boolean isWeightedGraph;
  ArrayList<Integer> _parentList;
  public Graph(boolean isDirected){
    _adjacencyMatrix = new ArrayList<ArrayList<Integer>>();
    _adjacencyList = new ArrayList<ArrayList<IntegerPair>>();
    _edgeList = new ArrayList<IntegerTriple>();
    _vertexMap = new HashMap<E, Integer>();
    this.isDirected = isDirected;
    _vertexList = new ArrayList<Vertex<E>>();
    _parentList = new ArrayList<Integer>();
  }
  public Graph(boolean isDirected, ArrayList<E> vertexItemList){
    this(isDirected);
    int size = vertexItemList.size();
    for(int i = 0; i<size; i++){
      _vertexMap.put(vertexItemList.get(i), UID);
      UID++;
      ArrayList<Integer> newList = new ArrayList<Integer>();
      for(int j = 0; j<size; j++){
        newList.add(0);
      }
      _adjacencyMatrix.add(newList);
      _adjacencyList.add(new ArrayList<IntegerPair>());
      _vertexList.add(new Vertex<E>(vertexItemList.get(i)));
      _parentList.add(-1);
    }
  }
  //TODO: initialise pq from edgelist
  public void addVertex(E e){
    Vertex<E> newVertex = new Vertex<E>(e);
    _vertexMap.put(e, UID);
    UID++;
    _adjacencyMatrix.add(new ArrayList<Integer>());
    resizeAdjacencyMatrix();
    _adjacencyList.add(new ArrayList<IntegerPair>());
    _vertexList.add(newVertex);
    _parentList.add(-1);
  }
  public void addEdge(E from, E to){
    addEdge(from, to, UNWEIGHTED);
  }
  public void addEdge(E from, E to, int weight){
    assert _vertexMap.containsKey(from);
    assert _vertexMap.containsKey(to);
    int fromIndex = _vertexMap.get(from);
    int toIndex = _vertexMap.get(to);
    _adjacencyMatrix.get(fromIndex).set(toIndex, weight);
    _adjacencyList.get(fromIndex).add(new IntegerPair(toIndex, weight)); //Possibly use Sorted Array to reduce access time
    if(!isDirected){
      _adjacencyMatrix.get(toIndex).set(fromIndex, weight);
      _adjacencyList.get(toIndex).add(new IntegerPair(fromIndex, weight));
    }
    _edgeList.add(new IntegerTriple(_vertexMap.get(from),_vertexMap.get(to),weight));
  }
  private void resizeAdjacencyMatrix(){
    int newLength = _adjacencyMatrix.size();
    for(int i = 0; i<newLength; i++){
      ArrayList<Integer> currList = _adjacencyMatrix.get(i);
      while(currList.size()<newLength){
        currList.add(NO_VERTEX);
      }
    }
  }
  //Query method: Breadth first search
  //O(V+E)
```

```java
public void BFS(int vertexIndex){
  assert vertexIndex <=_adjacencyList.size();
  resetParentList();
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  Queue<Integer> q = new LinkedList<Integer>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(false);
  }
  q.offer(vertexIndex);
  visited.set(vertexIndex,true);
  while(!q.isEmpty()){
    int currVertex = q.poll();
    System.out.println(currVertex);
    ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(currVertex);
    for(int i = 0; i<neighbourList.size(); i++){
      int neighbourIndex =neighbourList.get(i).getFirst();
      if(visited.get(neighbourIndex)==false){
        visited.set(neighbourIndex,true);
        _parentList.set(neighbourIndex,currVertex);
        q.offer(neighbourIndex);
      }
    }
  }
}
//Query method: Depth first search
//O(V+E)
public void DFS(int vertexIndex){
  resetParentList();
  assert vertexIndex <=_adjacencyList.size();
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(false);
  }
  DFS(vertexIndex,visited);
}
public void DFS(int vertexIndex, ArrayList<Boolean> visited){
  visited.set(vertexIndex, true);
  ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(vertexIndex);
  for(int i = 0; i<neighbourList.size();i++){
    int neighbourIndex =neighbourList.get(i).getFirst();
    if(visited.get(neighbourIndex)==false){
      visited.set(neighbourIndex,true);
      _parentList.set(neighbourIndex, vertexIndex);
      DFS(neighbourIndex, visited);
    }
  }
}
//getCutVertex modified from DFS
//O(V+E)
public ArrayList<Boolean> getCutVertex() {
  resetParentList();
    ArrayList<Boolean> visited = new ArrayList<Boolean>();
    ArrayList<Integer> timeOfFirstEncounter = new ArrayList<Integer>();
    ArrayList<Integer> timeOfFirstDiscovery = new ArrayList<Integer>();
    ArrayList<Boolean> cutVertex = new ArrayList<Boolean>();
    for(int i = 0; i<_adjacencyList.size(); i++){
      visited.add(false);
      timeOfFirstEncounter.add(-1);
      timeOfFirstDiscovery.add(-1);
      cutVertex.add(false);
    }
    int time = 0;
    getCutVertex(0, visited, timeOfFirstEncounter, timeOfFirstDiscovery, cutVertex,time);
    System.out.println(timeOfFirstDiscovery);
    System.out.println(timeOfFirstEncounter);
    return cutVertex;
  }
private void getCutVertex(int index, ArrayList<Boolean> visited, ArrayList<Integer> timeOfFirstEncounter,
                          ArrayList<Integer> timeOfFirstDiscovery, ArrayList<Boolean> cutVertex, int time){
    int numOfChildren = 0;
    visited.set(index, true);
    time++;
    timeOfFirstEncounter.set(index, time);
    timeOfFirstDiscovery.set(index, time);
    ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(index);
    for(int i = 0; i<neighbourList.size(); i++){
        int neighbourIndex = neighbourList.get(i).getFirst();
        if(visited.get(neighbourIndex)==false){
          numOfChildren++;
          _parentList.set(neighbourIndex, index);
          getCutVertex(neighbourIndex, visited, timeOfFirstEncounter, timeOfFirstDiscovery, cutVertex,time);
          timeOfFirstDiscovery.set(index, Math.min(timeOfFirstDiscovery.get(index), timeOfFirstDiscovery.get(neighbourIndex)));
          if(_parentList.get(index)==-1&&numOfChildren>1){
              cutVertex.set(index,true);
          }
          if(_parentList.get(index)!=-1&&timeOfFirstDiscovery.get(neighbourIndex)>=timeOfFirstEncounter.get(index)){
              cutVertex.set(index,true);
          }
        }else{
          if(neighbourIndex!=_parentList.get(index)){
              timeOfFirstDiscovery.set(index, Math.min(timeOfFirstDiscovery.get(index), timeOfFirstEncounter.get(neighbourIndex)));
          }
        }
    }
}
//Cycle Detection modified from DFS applied on Undirected Graph
//O(V+E)
public boolean hasCycle(){
  resetParentList();
  HashMap<Integer, Integer> backVertexMap = new HashMap<Integer,Integer>();
  return hasCycle(0, backVertexMap);
}
private boolean hasCycle(int vertexIndex, HashMap<Integer,Integer> backVertexMap){
  //System.out.println("Checking has cycle on vertex:" + vertexIndex);
  if(backVertexMap.containsKey(vertexIndex)){
    return true;
  }else{
    backVertexMap.put(vertexIndex, 1);
  }
  ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(vertexIndex);
  boolean ret = false;
  for(int i = 0; i<neighbourList.size();i++){
    int neighbourIndex =neighbourList.get(i).getFirst();
    if(hasCycle(neighbourIndex, backVertexMap)){
      return true;
    }
  }
  backVertexMap.remove(vertexIndex);
  return false;
}
//Path discovery modified from DFS
//O(V+E)
public ArrayList<Integer> findPath(int start, int end){
  resetParentList();
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(false);
  }
  Stack<Integer> pathStack = new Stack<Integer>();
  return findPath(start,end,visited, pathStack);
}
private ArrayList<Integer> findPath(int vertexIndex, int end, ArrayList<Boolean> visited, Stack<Integer> pathStack){
  //System.out.println("Finding path in vertex "+vertexIndex);
  visited.set(vertexIndex, true);
  pathStack.push(vertexIndex);
  if(vertexIndex==end){
    return new ArrayList<Integer>(pathStack);
  }
  ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(vertexIndex);
  for(int i = 0; i<neighbourList.size();i++){
    int neighbourIndex =neighbourList.get(i).getFirst();
    //System.out.println("recursing in vertex "+neighbourIndex);
    if(visited.get(neighbourIndex)==false){
      visited.set(neighbourIndex,true);
      _parentList.set(neighbourIndex, vertexIndex);
      ArrayList<Integer> a = findPath(neighbourIndex, end, visited, pathStack);
      if(a.size()!=0){
        return a;
      }
    }
  }
  pathStack.pop();
  return new ArrayList<Integer>();
}
//Topological Sort modified from DFS
//O(V+E)
public ArrayList<Integer> topologicalSort(){
  resetParentList();
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(false);
  }
  Stack<Integer> topologicalStack = new Stack<Integer>();
  for(int i = 0; i<visited.size();i++){
    if(visited.get(i)==false)
      topologicalSort(i,visited, topologicalStack);
  }
  ArrayList<Integer> topologicalOrder = new ArrayList<Integer>();
  while(!topologicalStack.empty()){
    topologicalOrder.add(topologicalStack.pop());
  }
  return topologicalOrder;
}
private void topologicalSort(int vertexIndex, ArrayList<Boolean> visited, Stack<Integer> topologicalStack){
  visited.set(vertexIndex, true);
  ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(vertexIndex);
  for(int i = 0; i<neighbourList.size();i++){
    int neighbourIndex =neighbourList.get(i).getFirst();
    if(visited.get(neighbourIndex)==false){
      visited.set(neighbourIndex,true);
      _parentList.set(neighbourIndex, vertexIndex);
      topologicalSort(neighbourIndex, visited, topologicalStack);
    }
  }
  topologicalStack.push(vertexIndex);
}
public ArrayList<Integer> kahnTopologicalSort(){
  ArrayList<Integer> inDegreeArray = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    //For each entry of inDegreeArray
    inDegreeArray.add(new Integer(0));
    for(int j = 0; j<_adjacencyList.size(); j++){
      if(_adjacencyMatrix.get(j).get(i)!=0){
        inDegreeArray.set(i, inDegreeArray.get(i)+1);
      }
    }
  }
  Queue<Integer> q = new LinkedList<Integer>();
  for(int i = 0; i<inDegreeArray.size(); i++){
    if(inDegreeArray.get(i)==0){
      q.add(i);
    }
  }
  ArrayList<Integer> topologicalOrder = new ArrayList<Integer>();
  while(!q.isEmpty()){
    int currVertex = q.poll();
    topologicalOrder.add(currVertex);
    ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
    for(int i = 0; i<neighbourList.size(); i++){
      IntegerPair currPair = neighbourList.get(i);
      int vertexTo = currPair.getFirst();
      int vertexToDegree = inDegreeArray.get(vertexTo);
      inDegreeArray.set(vertexTo, vertexToDegree-1);
      if(vertexToDegree-1==0){
        q.offer(vertexTo);
      }
    }
  }
  //System.out.println(inDegreeArray);
  return topologicalOrder;
```

```java
}
public int allTopologicalSort(){
  ArrayList<Integer> inDegreeArray = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    //For each entry of inDegreeArray
    inDegreeArray.add(new Integer(0));
    for(int j = 0; j<_adjacencyList.size(); j++){
      if(_adjacencyMatrix.get(j).get(i)!=0){
        inDegreeArray.set(i, inDegreeArray.get(i)+1);
      }
    }
  }
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  LinkedList<Integer> result = new LinkedList<Integer>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    visited.add(false);
  }
  return allTopologicalSortUtil(visited, result, inDegreeArray, 0);
}
private int allTopologicalSortUtil(ArrayList<Boolean> visited, LinkedList<Integer> result, ArrayList<Integer> inDegreeArray, int count){
  boolean flag = false;

  for(int i = 0; i<_adjacencyList.size(); i++){
    if(inDegreeArray.get(i)==0&&!visited.get(i)){
      visited.set(i, true);
      ArrayList<IntegerPair> neighbourList = _adjacencyList.get(i);
      for(int j = 0; j < neighbourList.size(); j++){
        IntegerPair currPair = neighbourList.get(j);
        int vertexTo = currPair.getFirst();
        int vertexToDegree = inDegreeArray.get(vertexTo);
        inDegreeArray.set(vertexTo, vertexToDegree-1);
      }
      result.add(i);
      count = allTopologicalSortUtil(visited, result, inDegreeArray, count);
      visited.set(i, false);
      result.removeLast();
      for(int j = 0; j < neighbourList.size(); j++){
        IntegerPair currPair = neighbourList.get(j);
        int vertexTo = currPair.getFirst();
        int vertexToDegree = inDegreeArray.get(vertexTo);
        inDegreeArray.set(vertexTo, vertexToDegree+1);
      }
      flag = true;
    }
  }
  if(!flag){
    System.out.println(result);
    count++;
  }
  return count;
}
//Count walks with exactly k edges from given source to given destination
//O(V^3)
int countWalksWithKEdges(int start, int end, int k){
  int numOfVertex = _adjacencyMatrix.size();
  int count[][][] = new int[numOfVertex][numOfVertex][k+1];
  for(int e = 0; e<=k;e++){
    for(int i = 0; i<numOfVertex; i++){
      for(int j = 0; j<numOfVertex; j++){
        count[i][j][e]=0;
        if(e==0&&i==j){
          count[i][j][e]=1;
        }
        if(e==1&&_adjacencyMatrix.get(i).get(j)!=0){
          count[i][j][e]=1;
        }
        if(e>1){
          for(int a = 0; a<numOfVertex;a++){
            if(_adjacencyMatrix.get(i).get(a)!=0){
              count[i][j][e]+=count[a][j][e-1];
            }
          }
        }
      }
    }
  }
  return count[start][end][k];
}
//Shortest Path given a DAG from Topological sorting
//O(V+E)
public void shortestPathinDAG(int source){
  ArrayList<Integer> topologicalOrder = topologicalSort();
  ArrayList<Integer> distance = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyMatrix.size();i++){
    distance.add(INF);
  }
  distance.set(source, 0);
  for(int i = 0; i<topologicalOrder.size();i++){
```

```java
    int currVertex = topologicalOrder.get(i);
    if(distance.get(currVertex)!=INF){
      ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
      for(int j = 0; j<neighbourList.size();j++){
        IntegerPair currPair = neighbourList.get(j);
        if(distance.get(currPair.getFirst())>distance.get(currVertex)+currPair.getSecond()){
          distance.set(currPair.getFirst(),distance.get(currVertex)+currPair.getSecond());
        }
      }
    }
  }
  for(int i = 0; i<_adjacencyMatrix.size();i++){
    if(distance.get(i)==INF){
      System.out.print("INF ");
    }else{
      System.out.print(distance.get(i)+" ");
    }
  }
}
//Check whether a graph is bipartite modified from DFS
//O(V+E)
public boolean isBipartite(){
  ArrayList<Integer> visited = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(-1);
  }
  return isBipartite(0, visited, 1);
}
private boolean isBipartite(int vertexIndex, ArrayList<Integer> visited, int color){
  visited.set(vertexIndex, color);
  ArrayList<IntegerPair> neighbourList = _adjacencyList.get(vertexIndex);
  boolean ret = true;
  for(int i = 0; i<neighbourList.size();i++){
    int neighbourIndex = neighbourList.get(i).getFirst();
    if(visited.get(neighbourIndex)==-1){
      ret = ret&&isBipartite(neighbourIndex, visited, 1-color);
      if(ret==false){
        return false;
      }
    }else{
      if(visited.get(neighbourIndex)+color!=1){
        return false;
      }
    }
  }
  return true;
}
//Find bridge in graph modified from DFS
//O(V+E)
public ArrayList<IntegerPair> getBridge() {
  resetParentList();
    ArrayList<Boolean> visited = new ArrayList<Boolean>();
    ArrayList<Integer> timeOfFirstEncounter = new ArrayList<Integer>();
    ArrayList<Integer> timeOfFirstDiscovery = new ArrayList<Integer>();
    ArrayList<IntegerPair> bridge = new ArrayList<IntegerPair>();
    for(int i = 0; i<_adjacencyList.size(); i++){
      visited.add(false);
      timeOfFirstEncounter.add(-1);
      timeOfFirstDiscovery.add(-1);
    }int time = 0;
    getBridge(0, visited, timeOfFirstEncounter, timeOfFirstDiscovery, bridge, time);
    return bridge;
  }
private void getBridge(int index, ArrayList<Boolean> visited, ArrayList<Integer> timeOfFirstEncounter,
                       ArrayList<Integer> timeOfFirstDiscovery, ArrayList<IntegerPair> bridge, int time){
    visited.set(index, true);
    time++;
    timeOfFirstEncounter.set(index, time);
    timeOfFirstDiscovery.set(index, time);
    ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(index);
    for(int i = 0; i<neighbourList.size(); i++){
        int neighbourIndex = neighbourList.get(i).getFirst();
        if(visited.get(neighbourIndex)==false){
          _parentList.set(neighbourIndex, index);
          getBridge(neighbourIndex, visited, timeOfFirstEncounter, timeOfFirstDiscovery, bridge,time);
          timeOfFirstDiscovery.set(index, Math.min(timeOfFirstDiscovery.get(index), timeOfFirstDiscovery.get(neighbourIndex)));
          if(timeOfFirstDiscovery.get(neighbourIndex)>timeOfFirstEncounter.get(index)){
              bridge.add(new IntegerPair(index, neighbourIndex));
            }
        }else{
          if(neighbourIndex!=_parentList.get(index)){
              timeOfFirstDiscovery.set(index, Math.min(timeOfFirstDiscovery.get(index), timeOfFirstEncounter.get(neighbourIndex)));
          }
        }
  }
}
//Prim algorithm for MST generation given a source vertex...return edge(with weight info)
//O(ElogV)
```

```java
public ArrayList<IntegerTriple> primMST(int source){
  ArrayList<Boolean> taken = new ArrayList<Boolean>();
  ArrayList<IntegerTriple> edgeInMST = new ArrayList<IntegerTriple>();
  PriorityQueue<IntegerTriple> edgeQueue = new PriorityQueue<IntegerTriple>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    taken.add(false);
  }
  process(source, edgeQueue, edgeInMST, taken);
  int mstWeight = 0;
  while(edgeQueue.isEmpty()==false){
    IntegerTriple leastWeight = edgeQueue.poll();
    if(taken.get(leastWeight.getThird())==false){
      mstWeight+=leastWeight.getFirst();
      edgeInMST.add(new IntegerTriple(leastWeight.getSecond(), leastWeight.getThird(), leastWeight.getFirst()));
      process(leastWeight.getThird(), edgeQueue, edgeInMST, taken);
    }
  }
  System.out.println(edgeInMST);
  System.out.println("Total cost: "+mstWeight);
  return edgeInMST;
}
private void process(int vertexIndex, PriorityQueue<IntegerTriple> edgeQueue, ArrayList<IntegerTriple> edgeInMST, ArrayList<Boolean> taken){
  taken.set(vertexIndex, true);
  ArrayList<IntegerPair> neighbourList = _adjacencyList.get(vertexIndex);
  for(int i = 0; i<neighbourList.size(); i++){
    IntegerPair weightedVector = neighbourList.get(i);
    if(taken.get(weightedVector.getFirst())==false){
      edgeQueue.offer(new IntegerTriple(weightedVector.getSecond(), vertexIndex, weightedVector.getFirst()));//weight, from, to
    }
  }
}
public ArrayList<IntegerTriple> kruskalMST(){
  sortEdgeListByWeight();
  UnionFind<IntegerTriple> edgeUnionFind = new UnionFind<IntegerTriple>(_edgeList);
  ArrayList<IntegerTriple> edgeInMST = new ArrayList<IntegerTriple>();
  int mstWeight = 0;
  for(int i = 0; i<_edgeList.size();i++){
    IntegerTriple currEdge = _edgeList.get(i);
    if(!edgeUnionFind.isSameSet(currEdge.getFirst(), currEdge.getSecond())){
      mstWeight+=currEdge.getThird();
      edgeInMST.add(currEdge);
      edgeUnionFind.unionSet(currEdge.getFirst(), currEdge.getSecond());
    }
  }
  System.out.println(edgeInMST);
  System.out.println("Total cost: "+mstWeight);
  return edgeInMST;
}
//Strongly connected components
//O(V+E)
public void printSCC(){
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  for(int i = 0; i<_adjacencyMatrix.size(); i++){
    visited.add(false);
  }
  Stack<Integer> s = new Stack<Integer>();
  for(int i = 0; i<_adjacencyMatrix.size(); i++){
    if(visited.get(i)==false){
      fillOrder(i, visited, s);
    }
  }
  int[][] transposedAdjacencyMatrix = new int[_adjacencyMatrix.size()][_adjacencyMatrix.size()];
  for(int i = 0; i<_adjacencyMatrix.size(); i++){
    for(int j=0; j<_adjacencyMatrix.size(); j++){
      transposedAdjacencyMatrix[i][j]=_adjacencyMatrix.get(j).get(i);
    }
  }
  visited = new ArrayList<Boolean>();
  for(int i = 0; i<_adjacencyMatrix.size(); i++){
    visited.add(false);
  }
  while(!s.empty()){
    int currVertex =s.pop();
    if(visited.get(currVertex)==false){
      DFSUtil(currVertex, visited, transposedAdjacencyMatrix);
      System.out.println();
    }
  }
}
private void fillOrder(int currVertex, ArrayList<Boolean> visited, Stack<Integer> s){
  visited.set(currVertex, true);
  ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
  for(int i = 0; i<neighbourList.size(); i++){
    IntegerPair currPair = neighbourList.get(i);
    int nextVertex = currPair.getFirst();
    if(!visited.get(nextVertex)){
      fillOrder(nextVertex, visited, s);
    }
```

```java
  }
  s.push(new Integer(currVertex));
}
private void DFSUtil(int currVertex, ArrayList<Boolean> visited, int[][] transposedAdjacencyMatrix){
  visited.set(currVertex, true);
  System.out.print(currVertex+" ");
  for(int i = 0; i<transposedAdjacencyMatrix.length; i++){
    if(transposedAdjacencyMatrix[currVertex][i]==0){
      continue;
    }
    if(visited.get(i)==false){
      DFSUtil(i, visited, transposedAdjacencyMatrix);
    }
  }
}
//PS: Strong Cut Vertex and Strong Bridges algorithms awaiting implementation

//Bellman Ford SSSP
//O(VE)
public ArrayList<Integer> BellmanFordSSSP(int source){
  int size = _adjacencyList.size();
  ArrayList<Integer> distanceArray = new ArrayList<Integer>();
  resetParentList();
  for(int i = 0; i<size;i++){
    distanceArray.add(INF);
  }
  distanceArray.set(source, 0);
  //System.out.println(_edgeList);
  for(int timeRelaxed = 0; timeRelaxed<size-1; timeRelaxed++){
    for(IntegerTriple currEdge: _edgeList){
      relax(currEdge.getFirst(), currEdge.getSecond(), currEdge.getThird(), distanceArray);
      if(!isDirected)
        relax(currEdge.getSecond(), currEdge.getFirst(), currEdge.getThird(), distanceArray);
    }
    //System.out.println(distanceArray);
  }
  boolean hasNegativeCycle = false;
  for(IntegerTriple currEdge:_edgeList){
    if(distanceArray.get(currEdge.getFirst())!=INF&&distanceArray.get(currEdge.getSecond())>
                                                    distanceArray.get(currEdge.getFirst())+currEdge.getThird()){
      hasNegativeCycle = true;
      System.out.println("Has negative Cycle, program terminated prematurely.");
      return new ArrayList<Integer>();
    }
  }
  return distanceArray;
}
private void relax(int from, int to, int weight, ArrayList<Integer> distanceArray){
  if(distanceArray.get(from)!=INF&&distanceArray.get(to)>distanceArray.get(from)+weight){
    distanceArray.set(to, distanceArray.get(from)+weight);
    _parentList.set(to, from);
  }
}
//O(V)
private ArrayList<Integer> backtrack(int dest, int source){
  if(_parentList.get(dest)==-1){
    return new ArrayList<Integer>();
  }else{
    ArrayList<Integer> ret = new ArrayList<Integer>();
    backtrack(dest, source, ret);
    return ret;
  }
}
private void backtrack(int dest, int source, ArrayList<Integer> path){
  int currVertex = dest;
  while(currVertex!=source){
    path.add(path.size(), currVertex);
    currVertex = _parentList.get(currVertex);
  }
  path.add(path.size(),source);
}
//Shortest Path Fast Algorithm
//O(VE) but O(E) for an random graph
public ArrayList<Integer> SPFA(int source){
  resetParentList();
  int size = _adjacencyList.size();
  ArrayList<Integer> distanceArray = new ArrayList<Integer>();
  resetParentList();
  for(int i = 0; i<size;i++){
    distanceArray.add(INF);
  }

  LinkedList<Integer> q = new LinkedList<Integer>();
  q.offer(source);
  distanceArray.set(source, 0);
  while(!q.isEmpty()){
    int currVertex = q.poll();
    //System.out.println(currVertex);
    ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
    for(int i = 0; i<neighbourList.size();i++){
      int neighbourIndex = neighbourList.get(i).getFirst();
      //System.out.println("Processing ("+currVertex+","+neighbourIndex+")");
      int weight = neighbourList.get(i).getSecond();
      relaxSPFA(currVertex, neighbourIndex, weight, distanceArray, q);
    }
  }
  return distanceArray;
}
private void relaxSPFA(int from, int to, int weight, ArrayList<Integer> distanceArray, LinkedList<Integer> queue){
  if(distanceArray.get(from)!=INF&&distanceArray.get(to)>distanceArray.get(from)+weight){
    distanceArray.set(to, distanceArray.get(from)+weight);
    _parentList.set(to, from);
    if(!queue.contains(to)){
      queue.offer(to);
      //System.out.println(queue);
    }
  }
}
//SSSP for unweighted graph: BFS//Tree
//O(V+E)
public ArrayList<Integer> SSSP_BPS(int source){
  ArrayList<Integer> distanceArray = new ArrayList<Integer>();
  resetParentList();
  for(int i = 0; i<_adjacencyList.size();i++){
    distanceArray.add(INF);
  }
  distanceArray.set(source, 0);
  ArrayList<Boolean> visited = new ArrayList<Boolean>();
  Queue<IntegerPair> q = new LinkedList<IntegerPair>();
  for(int i = 0; i<_adjacencyList.size();i++){
    visited.add(false);
  }
  q.offer(new IntegerPair(source, 0));
  visited.set(source,true);
  while(!q.isEmpty()){
    IntegerPair p= q.poll();
    int currVertex = p.getFirst();
    int currLayer = p.getSecond();
    ArrayList<IntegerPair> neighbourList =  _adjacencyList.get(currVertex);
    for(int i = 0; i<neighbourList.size(); i++){
      int neighbourIndex =neighbourList.get(i).getFirst();
      if(visited.get(neighbourIndex)==false){
        visited.set(neighbourIndex,true);
        _parentList.set(neighbourIndex,currVertex);
        distanceArray.set(neighbourIndex, currLayer+1);
        q.offer(new IntegerPair(neighbourIndex, currLayer+1));
      }
    }
  }
  return distanceArray;
}

public ArrayList<Integer> SSSP_DAG(int source){
  ArrayList<Integer> topologicalOrder = this.topologicalSort();
  ArrayList<Integer> distanceArray = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    distanceArray.add(INF);
  }
  resetParentList();
  distanceArray.set(topologicalOrder.get(0), 0);
  for(int i = 0; i<topologicalOrder.size();i++){
    int currVertex = topologicalOrder.get(i);
    ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
    for(int j =  0; j<neighbourList.size(); j++){
      IntegerPair currPair = neighbourList.get(j);
      int toVertex = currPair.getFirst();
      int weight = currPair.getSecond();
      if(distanceArray.get(currVertex)!=INF&&distanceArray.get(currVertex)+weight<distanceArray.get(toVertex)){
        distanceArray.set(toVertex, distanceArray.get(currVertex)+weight);
        _parentList.set(toVertex, currVertex);
      }
    }
  }
  return distanceArray;
}
/*
//Original Dijkstra
public ArrayList<Integer> dijkstra_original(int source){

}
*/
//Modified Dijkstra
public ArrayList<Integer> dijkstra_modified(int source){
  ArrayList<Integer> distanceArray = new ArrayList<Integer>();
  for(int i = 0; i<_adjacencyList.size(); i++){
    distanceArray.add(INF);
  }
  resetParentList();
  distanceArray.set(source, 0);
  PriorityQueue<IntegerPair> pq = new PriorityQueue<IntegerPair>();
  pq.offer(new IntegerPair(0, source));
  while(!pq.isEmpty()){
    IntegerPair currPair = pq.poll();
    int currVertex = currPair.getSecond();
    int cost= currPair.getFirst();
    if(distanceArray.get(currVertex)==cost){
      ArrayList<IntegerPair> neighbourList = _adjacencyList.get(currVertex);
      for(int i = 0; i<neighbourList.size(); i++){
        IntegerPair p = neighbourList.get(i);
        int toVertex = p.getFirst();
        int weight = p.getSecond();
        if(distanceArray.get(currVertex)!=INF&&distanceArray.get(toVertex)>distanceArray.get(currVertex)+weight){
          distanceArray.set(toVertex, distanceArray.get(currVertex)+weight);
          _parentList.set(toVertex, currVertex);
          pq.offer(new IntegerPair(distanceArray.get(toVertex), toVertex));
        }
      }
    }
  }
  return distanceArray;
}
//Floyd Warshall
//O(V^3)
public int[][] floydWarshall(){
  int numOfVertex = _adjacencyList.size();
  int[][] ret = new int[numOfVertex][numOfVertex];
  for(int i = 0; i < numOfVertex; i++){
    for(int j = 0; j< numOfVertex; j++){
      if(i==j){
        ret[i][j]=0;
      }else{
        ret[i][j]=_adjacencyMatrix.get(i).get(j)==0?INF:_adjacencyMatrix.get(i).get(j);
      }
    }
  }

  for(int k = 0; k<numOfVertex; k++){
    for(int i = 0; i< numOfVertex; i++){
      for(int j = 0; j<numOfVertex; j++){
        if(ret[i][k]!=INF&&ret[k][j]!=INF&&ret[i][k]+ret[k][j]<ret[i][j]){
          ret[i][j]=ret[i][k]+ret[k][j];
        }
      }
    }
  }
  return ret;
}
//Shortest Path from u to v with k edges
public int shortestPathWithKEdges(int source, int dest, int k){
  int numOfVertex = _adjacencyMatrix.size();
  int sp[][][] = new int[numOfVertex][numOfVertex][k+1];
  for(int e = 0; e<=k; e++){
    for(int i = 0; i< numOfVertex; i++){
      for(int j = 0; j<numOfVertex; j++){
        sp[i][j][e]=INF;
        if(e==0&&i==j){
          sp[i][j][e]=0;
        }
        if(e==1&&_adjacencyMatrix.get(i).get(j)!=0){
          sp[i][j][e]=_adjacencyMatrix.get(i).get(j);
        }
        if(e>1){
          for(int a = 0; a< numOfVertex; a++){
            if(_adjacencyMatrix.get(i).get(a)!=INF&&i!=a&&j!=a&&sp[a][j][e-1]!=INF){
              sp[i][j][e]=Math.min(sp[i][j][e], _adjacencyMatrix.get(i).get(a)+sp[a][j][e-1]);
            }
          }
        }
      }
    }
  }
  return sp[source][dest][k];
}

//Auxilliary method
//O(V)
public void resetParentList(){
  for(int i = 0; i<_parentList.size();i++){
    _parentList.set(i,-1);
  }
}
public void sortAdjacencyList(){
  for(int i = 0; i <_adjacencyList.size();i++){
    ArrayList<IntegerPair> currList = _adjacencyList.get(i);
```

```
      Collections.sort(currList);
    }
  }
  public void sortEdgeList(){
    Collections.sort(_edgeList);
  }
  public void sortEdgeListByWeight(){
    final Comparator<IntegerTriple> weightAsendingOrder = new Comparator<IntegerTriple>(){
      public int compare(IntegerTriple o1, IntegerTriple o2){
        return o1.getThird()-o2.getThird();
      }
    };
    Collections.sort(_edgeList, weightAsendingOrder);
  }
}
```