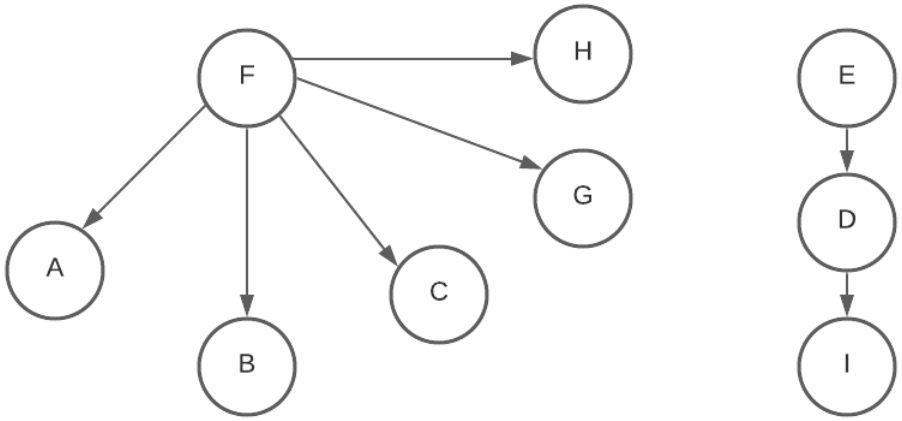
Lab assignment solution

Problem 1:

A. The graph G is not connected because there is no path between B and D. there are two components.

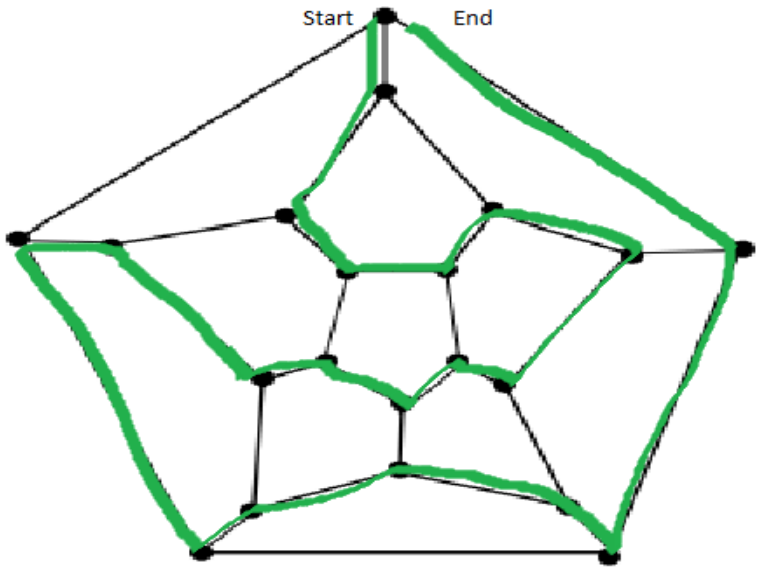
B.



C. It is not Hamiltonian graph because there can not be a cycle that spans all the vertices. If a cycle has vertex 'F' in it, then it cannot have vertex 'E', because they're on a separate component.

D. Yes, $C = \{F, A, G, D, E\}$

Problem 2:



Problem 3:

```
// Smallest vertex cover
Algorithm smallestVertexCover(V, E)
    Input: V, a set of all vertex and E, a set of all edges
    Output: the size of a smallest possible vertex cover
    allSubsets = powerset(V)
    for subset in allSubsets
        coversAll ← true
        for e in edges
            (v1,v2) ← computeEndpoints(e)
            if(!belongsTo(v1,subset) && !belongsTo(v2,subset))
                coversAll ← false
                break
        if(coversAll)
            return subset.size()

Algorithm powerset(S)
    Input: a set S of n elements
    Output: the power set of S
    ps ← a set containing an empty set
    while(!S.isEmpty())
        a ← S.removeOne()
        for(x in ps)
            temp ← x U a
        ps ← ps U temp
    return ps
```

Problem 4:

IsConnected.java

```java
package lab11_undirected;

public class IsConnected extends BreadthFirstSearch {
    int numberOfComponents = 0;
    public IsConnected(Graph graph) {
        super(graph);
    }
    @Override
    protected void additionalProcessing() {
        numberOfComponents++;
    }

    //TO-DO
    public boolean isConnected() {
        start();
        return numberOfComponents == 1;
    }
}

```

HasCycle.java

```java
package lab11_undirected;


public class HasCycle extends BreadthFirstSearch {

    public HasCycle(Graph graph) {
        super(graph);
    }

    //TO-DO
    public boolean hasCycle() {
        int numberOfEdgesInGraph = graph.edges().size();
        FindSpanningTree fst = new FindSpanningTree(graph);
        fst.computeSpanningTree();
        int numberOfEdgesInTree = fst.getTreeSize();
        return numberOfEdgesInTree < numberOfEdgesInGraph;
    }
}

```

PathExists.java

```java

public class PathExists extends BreadthFirstSearch {
    boolean pathFound = false;
    int componentCount = 0;
    Vertex v;
    public PathExists(Graph graph) {
        super(graph);
    }
    @Override
    protected void additionalProcessing() {
        componentCount++;
    }
    @Override
    protected void processVertex(Vertex v) {
//      System.out.println("found -- ");
        if(this.v.equals(v) && componentCount==0)
            pathFound = true;
    }

    //TO-DO
    public boolean pathExists(Vertex u, Vertex v) {
        pathFound = false;
        this.v = v;
        start(u);
        return pathFound;
    }

}
```

ShortestPath.java

```java
public class ShortestPath extends BreadthFirstSearch {
    private HashMap<Vertex, Integer> levelsMap = new HashMap<Vertex, Integer>();
    private HashMap<Vertex, Vertex> parentMap = new HashMap<Vertex, Vertex>();

    /** Assumes g is connected */
    public ShortestPath(Graph g) {
        super(g);
    }

    @Override
    protected void processEdge(Edge e) {
        if(parentMap.containsKey(e.v) && !parentMap.containsKey(e.u)) {
            levelsMap.put(e.u, levelsMap.get(e.v)+1);
            parentMap.put(e.u, e.u);
        }

    }
    public int getLevel(Vertex v) {
        return levelsMap.get(v);
    }
    //TO-DO
    public int computeShortestPathLength(Vertex s, Vertex v) {
        parentMap.put(s, s);
        levelsMap.put(s, 0);
        start(s);
        return levelsMap.get(v);
    }
}
```

Graph.java

```java
    //TO-DO
    public boolean isConnected() {
        IsConnected ic = new IsConnected(this);
        return ic.isConnected();
    }
    //TO-DO
    public boolean hasPathBetween(Vertex u, Vertex v) {
        PathExists pe = new PathExists(this);
        return pe.pathExists(u, v);
    }
    //TO-DO
    public boolean containsCycle() {
        HasCycle hc = new HasCycle(this);
        return hc.hasCycle();
    }
    //TO-DO
    int shortestPathLength(Vertex u, Vertex v) {
        ShortestPath sp = new ShortestPath(this);
        return sp.computeShortestPathLength(u, v);
    }
```