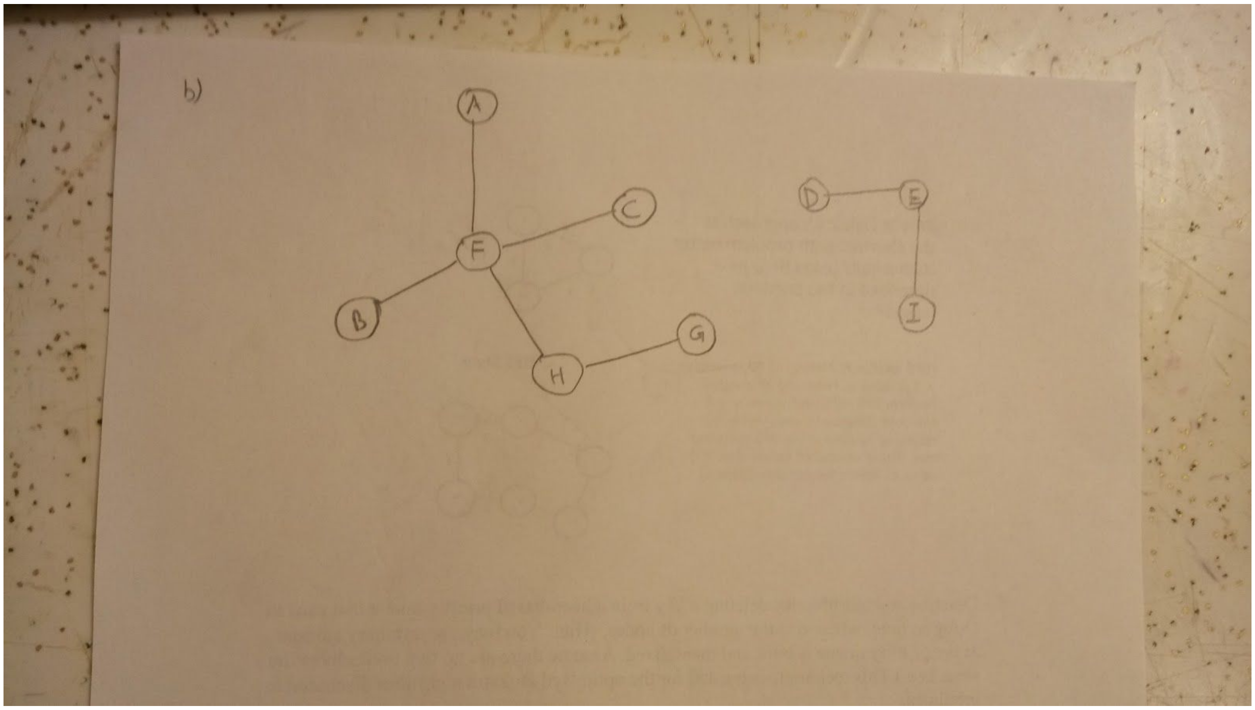
Problem 1

a) The graph is not connected. It has 2 connected components

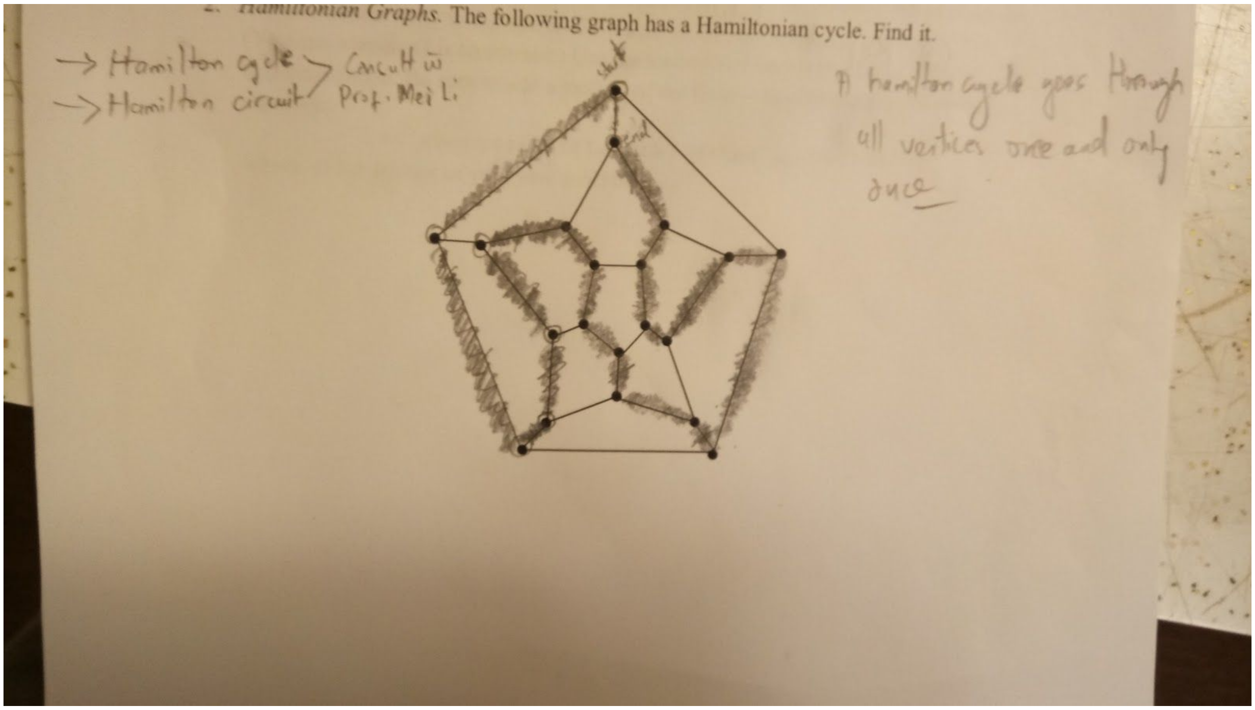
b) Spanning forest



c) The graph cannot have a hamilton cycle because it's disconnected. Note that a hamilton cycle goes through all of vertices once.

d) A vertex cover is a set of all vertices that cover all the edges . In our case of course there exists more than one vertex covers like {F,A,G,E,D}

Problem 2



Problem 3

```
Algorithm: SmallestVertexCover
    Input: A graph G whose set of vertices is denoted V and set of edges is denoted E
    Output: Smallest size of a vertex cover U for G
    pow ← PowerSet(V)
    minCover ← V
    minVal ← |V|
    for each U in pow do
        isCover ← true
        //verify U is a vertex cover
        for each e in E do
            (u,v) ← computeEndpoints(e)
            if( !(belongsTo(u,U) and !belongsTo(v,U))
                isCover ← false
            if(isCover and U.size() < minCover.size()) then
                minCover ← U
                minVal ← |U|
    return minVal
```

Problem 4

```
//Checking if there is a path btn 2 vertices

public class PathExists extends BreadthFirstSearch {
	private Vertex target;
	private boolean pathFound = false;
	private int numComponents = 0;
	public PathExists(Graph graph) {
		super(graph);
	}
	@Override
	public void processVertex(Vertex w) {
		//Change value of pathFound only if we are working in
		//the 0th component.
		if(w.equals(target) && numComponents == 0) pathFound = true;
	}
	@Override
	public void additionalProcessing() {
		numComponents++;
	}
	public boolean pathExists(Vertex u, Vertex v) {
		target = v;
		start(u);
		return pathFound;
	}
}

//Checking if the graph is connected and how many components exist
public class IsConnected extends BreadthFirstSearch {
	private int numComponents = 0;
	public IsConnected(Graph graph) {
		super(graph);
	}

	@Override
	public void additionalProcessing() {
		numComponents++;
	}

	public boolean isConnected() {
		start();
		return numComponents == 1;
```

```
	}

}
//Checking if there is a cycle

public class HasCycle extends BreadthFirstSearch {
//	private ArrayList<Edge> tree = new ArrayList<Edge>();
	private int numTreeEdges = 0;
	private int numGraphEdges = 0;
	public HasCycle(Graph graph) {
		super(graph);
		numGraphEdges = graph.edges().size();
	}
	protected void processEdge(Edge e) {
			//tree.add(e);
			++numTreeEdges;
	}

	public boolean hasCycle() {
		start();
		return numGraphEdges > numTreeEdges;
	}

}
```

Problem 5

```
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);
	}
	protected void processVertex(Vertex v) {
		Vertex parent = parentMap.get(v);
		if(parent == null) // v has no parent, v is the starting vertex
			levelsMap.put(v, 0);
		else
			levelsMap.put(v, levelsMap.get(parent) + 1);
```

```
	}
	@Override
	protected void processEdge(Edge e) {
		//first component is child, second component is parent
		parentMap.put(e.u, e.v);
	}
	public int computeShortestPathLength(Vertex s, Vertex v) {
		start(s);
		//now levels and parents have been computed
		return levelsMap.get(v);
	}

	public List<Edge> computeShortestPath(Vertex s, Vertex v) {
		start(s);
		//now levels and parents have been computed
		return shortestPath(new ArrayList<Edge>(), s, v);

	}

	private List<Edge> shortestPath(List<Edge> temp, Vertex s, Vertex v) {
		if(v.equals(s)) {
			return temp;
		}
		Vertex w = parentMap.get(v);
		temp.add(0, new Edge(w, v)); //add to the front of the list
		return shortestPath(temp, s, w);
	}

}
```