**Algorithm: Lab11 (By Sujiv Shrestha ID:610145)**

**Problem 1.**

1. Answer questions about the graph G = (V,E) displayed below.



A. Is the graph G connected? If not, what are the connected components for G?

Answer: The graph G is disconnected. The connected components for G are G1 = ({D,E,I}, {D-E,D-I,I-E}) and G2 = ({B,A,C,F,G,H},{B-A, A-C, A-F, B-F, F-C, F-H, C-G, G-H})

B. Draw a spanning tree/forest for G.

C. Is G a Hamiltonian graph?

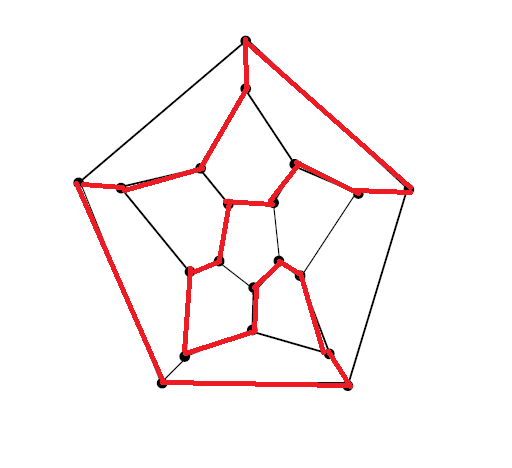
Answer: G is not a Hamiltonian graph because in Hamiltonian graph there should be atleast one cyclic path (Hamiltonian cycle) connecting all the vertices in that graph but here the graph is disconnected.

D. Is there a Vertex Cover of size less than or equal to 5 for G? If so, what is the Vertex Cover?

Answer: Yes, there is a Vertex Cover of size less than or equal to 5 for G. That Vertex Cover is V={D,E,F,G,A}

**Problem 2.**

2. *Hamiltonian Graphs.* The following graph has a Hamiltonian cycle. Find it.

****

**Problem 3.**

3. *Vertex Covers.* Create an algorithm for computing the smallest size of a vertex cover for a graph. The input of your algorithm is a set V of vertices along with a set E of edges. Assume you have the following functions available (no need to implement these):

* computeEndpoints(edge) – returns the vertices that are at the endpoints of the input edge
* belongsTo(vertex, set) – returns true if the input vertex is a member of the given set

*Hint:* Loop through all subsets of V. For each subset W, check to see if W is a vertex cover. Do this by looping through all edges; for each edge e, check to see if at least one of its endpoints lies in W.

Solution:

**Algorithm** vertexCover(V, E)

**Input** set of vertex V and set of Edges E

**Output** vertex Cover VC

VC←emptyList of vertex

**while** (¬E.isEmpty()) **do**

E1 ← E.removeFirst()

(V1,V2) ←computeEndpoints(E1)

**if**(¬belongsTo(V1, VC) **and** ¬belongsTo(V2, VC)) **then**

VC.insertLast(V2)

**return** VC

**Problem 4.**

4. Compute two spanning trees for the graphs below using algorithms we discuss in class. (You can start with vertex A) Are the two spanning trees same?



Solution:

1. Spanning tree (Depth-First Search algorithm)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | |  | |  | |  | |  | | A | | |  | | --- | |  | |  | |  | |  | | F | | A |   {A-F} |
| |  | | --- | |  | |  | |  | | E | | ~~F~~ | | A |   {A-F,F-E} | |  | | --- | |  | |  | | D | | ~~E~~ | | ~~F~~ | | A |   {A-F,F-E,E-D} |
| |  | | --- | |  | | C | | D | | ~~E~~ | | ~~F~~ | | A |   {A-F,F-E,E-D,D-C} | |  | | --- | | ~~B~~ | | ~~C~~ | | ~~D~~ | | ~~E~~ | | ~~F~~ | | ~~A~~ |   {A-F,F-E,E-D,D-C,C-B} |

1. Spanning tree (Breadth-First Search Algorithm)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | |  | |  | |  | |  | | A | | |  | | --- | |  | | B | | D | | E | | F | | ~~A~~ |   {A-F, A-E, A-D, A-B} |
| |  | | --- | |  | | B | | D | | ~~E~~ | | ~~F~~ | | ~~A~~ |   {A-F, A-E, A-D, A-B} | |  | | --- | | ~~C~~ | | ~~B~~ | | ~~D~~ | | ~~E~~ | | ~~F~~ | | ~~A~~ |   {A-F, A-E, A-D, A-B, D-C} |

**Problem 5.**

5. Write the pesudo-code for compute connected components algorithm discussed in class. Your algorithm can be built on top of DFS discussed in the slides.

Solution:

**Algorithm**: AbstractGraphSearch

**while** (some vertex not yet visited) **do**

handleInitialVertext()

singleComponentLoop()

additionalProcessing()

**Algorithm**: additionalProcessing

Increase currentComponentNumber by 1

**Algorithm**: processVertex(Vertex v)

**Input**: Vertex v of a graph

componentMap[currentComponentNumber].insertLast(v)

vertexComponentMap.add(v,currentComponentNumber)

**Algorithm**: computeConnectedComponents

**Input**: //Nothing

**Output**: total number of connected components in the graph

AbstractGraphSearch.start()

**return** currentComponentNumber

Java Implementation

**public** **class** ConnectedComponentSearch **extends** DepthFirstSearch {

ArrayList<List<Vertex>> componentMap = **new** ArrayList<>();

HashMap<Vertex,Integer> vertexComponentMap = **new** HashMap<>();

**int** currentComponentNumber = 0;

**public** ConnectedComponentSearch(Graph graph) {

**super**(graph);

}

@Override

**protected** **void** handleInitialVertex() {

**if**(currentComponentNumber<=componentMap.size())

componentMap.add(**new** ArrayList<>());

**super**.handleInitialVertex();

}

@Override

**public** **void** processVertex(Vertex v) {

//super.processVertex(v);

componentMap.get(currentComponentNumber).add(v);

vertexComponentMap.put(v, currentComponentNumber);

}

@Override

**public** **void** additionalProcessing() {

**super**.additionalProcessing();

//if(someVertexUnvisited()) {

currentComponentNumber++;

//componentMap.add(new ArrayList<>());

//System.out.println(someVertexUnvisited());

//}

}

**public** **int** computeConnectedComponent() {

**super**.start();

**return** currentComponentNumber;

}

}

//inside main() of Main.java

List<Pair> l = **new** ArrayList<Pair>();

l.add(**new** Pair("A","B"));

l.add(**new** Pair("B","C"));

l.add(**new** Pair("A","D"));

l.add(**new** Pair("B","D"));

l.add(**new** Pair("E", "F"));

Graph g = **new** Graph(l);

ConnectedComponentSearch c = **new** ConnectedComponentSearch(g);

System.***out***.println("Total Components:"+c.computeConnectedComponent());

System.***out***.println(c.componentMap);

System.***out***.println(c.vertexComponentMap);

Output:

Total Components:2

[[A, B, C, D], [E, F]]

{A=0, B=0, C=0, D=0, E=1, F=1}

**Problem 6.**

6. Write the pesudo-code for the algorithm, discussed in class, that computes the shortest path length between two vertices in a graph. You can assume that:

a. The graph is connected.

b. A version of BFS is provided that accepts a specified starting vertex.

Solution:

**Algorithm**: AbstractGraphSearch

**while** (some vertex not yet visited) **do**

handleInitialVertext()

singleComponentLoop()

additionalProcessing()

**Algorithm**: handleInitialVertex

parentMap.add(start,start)

levelsMap.add(start,0)

queue.add(start)

mark start as visited

**Algorithm**: processVertex(Vertex v)

**Input**: Vertex v of a graph

parent← parentMap.get(v)

levelsMap.add(v, levelsMap.get(parent)+1)

**Algorithm**: processEdge(Edge edge)

**Input**: Edge edge of a graph

parentMap.add(edge.v, edge.u)

**Algorithm**: computeShortestPath

**Input**: start point Vertex ‘start’ and end point Vertex ‘end’ of path

**Output**: length of the shortest possible path from start to end

shortPath←empty list of Vertex

AbstractGraphSearch.start()

traversal←end

**While**(¬parentMap.get(traversal)=traversal) **do**

shortPath.insertFirst(traversal)

traversal←parentMap.get(traversal)

shortPath.insertFirst(start)

**return** shortPath.size() -1

Java Implementation

**public** **class** ShortPathLength **extends** BreadthFirstSearch {

HashMap<Vertex,Integer> levelsMap = **new** HashMap<>();

HashMap<Vertex,Vertex> parentMap = **new** HashMap<>();

Vertex start, end;

**public** ShortPathLength(Graph g) {

**super**(g);

}

**public** List<Vertex> computeShortestPath(Vertex s, Vertex e){

start = s;

end = e;

Vertex trav = e;

**super**.start();

List<Vertex> shortPath = **new** ArrayList<Vertex>();

System.***out***.println(parentMap);

System.***out***.println(levelsMap);

**while**(parentMap.get(trav)!=trav) {

shortPath.add(0,trav);

trav = parentMap.get(trav);

}

shortPath.add(0,start);

System.***out***.println("Shortest path length is:"+levelsMap.get(end));

**return** shortPath;

}

@Override

**protected** **void** handleInitialVertex() {

setHasBeenVisited(start);

levelsMap.put(start, 0);

parentMap.put(start,start);

queue.add(start);

}

@Override

**protected** **void** singleComponentLoop() {

**while**(!queue.isEmpty()){

Vertex v = nextUnvisitedAdjacent(queue.peek());

**while**(v!=**null**) {

System.***out***.println("Vertex: "+v);

setHasBeenVisited(v);

processEdge(**new** Edge(queue.peek(),v));

processVertex(v);

queue.add(v);

v=nextUnvisitedAdjacent(queue.peek());

System.***out***.println("Queue: "+queue);

}

queue.remove();

}

}

@Override

**protected** **void** processVertex(Vertex w){

levelsMap.put(w, levelsMap.get(parentMap.get(w))+1);

}

@Override

**protected** **void** processEdge(Edge edge) {

parentMap.put(edge.v, edge.u);

}

}

//In main() of Main.java

List<Pair> l = **new** ArrayList<Pair>();

l.add(**new** Pair("A","B"));

l.add(**new** Pair("B","C"));

l.add(**new** Pair("A","D"));

l.add(**new** Pair("B","D"));

l.add(**new** Pair("D","E"));

l.add(**new** Pair("E", "F"));

Graph g = **new** Graph(l);

ShortPathLength spl = **new** ShortPathLength(g);

System.***out***.println("Short Path is:"+spl.computeShortestPath(**new** Vertex("C"), **new** Vertex("F")));

Output:

Shortest path length is:4

Short Path is:[C, B, D, E, F]