**Programming Assignment Graphs CS-254**

**Breadth and Depth First Searches**

Goals: Advance skills in implementing pseudocode. Learn more about the Java Queue, Stack and HashMap as the data structures used in the implementation.

1. Create a new Java project in eclipse.
2. Create the following classes from the UML
3. Add a GraphSearchDriver with a main method. Use the provided code to test your implementations. Comment out most of the code and then add it little by little to test it.
4. Implement your toString methods to match the provided output. Note that the trees could be slightly different depending on your search.

|  |
| --- |
| UndirectedGraph |
| * vertices: HashMap<String, LinkedList<Vertex>> * listOfVertices: ArrayList<Vertex> |
| + UndirectedGraph(ArrayList<Vertex> listOfVertices): //constructor  + addEdge(Vertex u, Vertex v):boolean  + breadthFirstSearch(Vertex start): UndirectedGraph //the BFS tree  + depthFirstSearch(Vertex start): UndirectedGraph //the DFS tree  + toString():String |

**UndirectedGraph UML Details:**

**Instance Variables:**

**vertices** – instance variable to store the name of a Vertex u as the key and its LinkedList of adjacent vertices to u as the value in a HashMap.

**listOfVertices** – an ArrayList of all the vertices in the UndirectedGraph.

**Methods:**

**UndirectedGraph(ArrayList<Vertex> listOfVertices) -** Create the HashMap in the constructor and put each Vertex in vertices and an empty LinkedList<Vertex> into the hashmap. (The edges will be added through the addEdge method)

Assign to listOfVertices the instance variable of same name:

**this**.listOfVertices = listOfVertices;

**addEdge(Vertex u, Vertex v)** – As this in an undirected graph, add v to the adjacency list for u and add u to the adjacency list for v. Be sure to include both directions.

**breadthFirstSearch**(Vertex start) and **depthFirstSearch**(Vertex start) – implement algorithms listed later in this document.

**toString()** - Print out each vertex and its adjacency list to match the provided output.

Full graph1:

Adjacency list for graph:

Vertex A: ->B

Vertex B: ->A ->C ->D

Vertex C: ->B ->D

Vertex D: ->B ->C

|  |
| --- |
| Vertex |
| * name: String * color: int * visited: boolean   + WHITE: int //1  + GREEN: int //2  + BLACK: int //3 |
| + Vertex(name: String): //constructor  + getName():String  + getColor():int  + setColor(color: int):void  + isVisited(): boolean  + setVisited(visited: boolean): void  + toString():String |

**Vertex UML Details:**

**Instance Variables:**

**name** – name of the Vertex

**color** – color of Vertex. 1 is WHITE, 2 is GREEN, 3 is BLACK. Match the constants.

**visited** – store if the vertex has been visited.

**Methods:**

**getters and setters** – As usual.

**toString()** - return the vertex name.

**Your code must implement the algorithms listed below. You must comment your code with exactly what part of the algorithm it is implementing. You must list Input, Output, and each line of the algorithm in comments by the code that implements it. Every method should have documentation in Javadocs. Documentation 20 points. No credit for code that does not have algorithm comments to match code.**

ALGORITHM Iterative *BreadthFirstSearch*(Graph G, Vertex start)

//Traverses a graph using the breadth first methodology

//Input: A Graph G that is a connected and undirected graph. //Vertex start is the vertex to start the traversal.

//Output: Graph tree that represents the breadth first search tree traversal.

**BreadthFirstSearch( Vertex start) {**

initialize vertices; // mark all vertices as white

Create Graph *T (BFS tree)* using all vertices from G

Q = {Start}; *// Q is a queue; insert vertex u*

**while** (Q not empty) {

u = Dequeue Q

**for** each v ∈ u->adjacent **do**{

**if** (v->color == WHITE) {

v->color = GREEN;

Enqueue to Q Vertex v

tree *T* add edge(u,v)

}//end if

}//**end** for each

u->color = BLACK;

}//**end** while

}//**end** algorithm

In depth-first search the idea is to travel as deep as possible from neighbor to neighbor before backtracking. What determines how deep is possible is that you must follow edges, and you don't visit any vertex twice. To do this properly we need to keep track of which vertices have already been visited, plus how we got to (the path to...) where we currently are, so that we can backtrack.

ALGORITHM Iterative *DepthFirstSearch*( Vertex start)

//Traverses a graph using the depth first methodology

//Input: A graph that is a connected and undirected.

//Vertex start is the vertex to start the traversal.

//Output: Graph tree that represents the depth first search tree.

**DepthFirstSearch(Vertex start) {**

create Stack S

create ArrayList<Vertex> visited //store order of visited vertices

create HashMap<Vertex,Vertex> predecessor //store predecessor value

initialize each vertex u in the graph to set visited false

push Vertex start onto S

**while** S is not empty **do**

u := pop S;

**if** not visited u **then**

set visited u = true

add u to visited

**for** each unvisited neighbor w of u **do**

**if**(w not visited)**do**

push Vertex w onto S

put w as key and u as value predecessor Hashmap

**//end if**

**//end for**

//**end if**

**//end while**

Create Undirected Graph tree with vertices visited

**for** each Vertex v stored in visited **do**

**if** predecessor contains key for v

get Vertex u from predecessor

add edge (u, v) to tree

**//end if**

**//end for**

return tree

**//end DFS**

**Output from provided driver code:**

Vertices only:

Adjacency list for graph:

Vertex A:

Vertex B:

Vertex C:

Vertex D:

Full graph1:

Adjacency list for graph:

Vertex A: ->B

Vertex B: ->A ->C ->D

Vertex C: ->B ->D

Vertex D: ->B ->C

BFS Tree:

Adjacency list for graph:

Vertex A: ->B

Vertex B: ->A ->C ->D

Vertex C: ->B

Vertex D: ->B

DFS Tree:

Adjacency list for graph:

Vertex A: ->B

Vertex B: ->A ->D

Vertex C: ->D

Vertex D: ->B ->C

Graph2:

Adjacency list for graph:

Vertex a: ->c ->d

Vertex b: ->c ->f

Vertex c: ->a ->e ->b

Vertex d: ->a ->e

Vertex e: ->c ->d ->f

Vertex f: ->e ->b

BFS Tree2 from a:

Adjacency list for graph:

Vertex a: ->c ->d

Vertex b: ->c

Vertex c: ->a ->e ->b

Vertex d: ->a

Vertex e: ->c ->f

Vertex f: ->e

DFS Tree2 from a:

Adjacency list for graph:

Vertex a: ->d

Vertex b: ->f ->c

Vertex c: ->b

Vertex d: ->a ->e

Vertex e: ->d ->f

Vertex f: ->e ->b

BFS Tree2 from c:

Adjacency list for graph:

Vertex a: ->c ->d

Vertex b: ->c

Vertex c: ->a ->e ->b

Vertex d: ->a

Vertex e: ->c ->f

Vertex f: ->e

DFS Tree2 from c:

Adjacency list for graph:

Vertex a: ->d

Vertex b: ->c ->f

Vertex c: ->b

Vertex d: ->e ->a

Vertex e: ->f ->d

Vertex f: ->b ->e