Muhammad Ansari

December 12th, 2017

COMP 482 Algorithm Design

Project #3

In implementing a isStronglyConnected method, I researched significantly into common algorithms used to perform this task and stumbled upon the widely-used Kosaraju’s algorithm. This algorithm involves using depth-first-search on a graph and then on the graph’s transpose in order to determine whether all nodes are connected to one another or not. The actual steps involved in this algorithm are:

1. Initializing all vertices as not-visited
2. Performing depth-first-search on the graph from an arbitrary starting vertex. If the traversal visits all of the vertices of the graph, then return true, else return false.
3. Transposing the graph.
4. Initializing all vertices as not-visited.
5. Performing depth-first-search on the reversed graph from the same starting vertex. If the traversal visits every vertex, then return true, else false.

In implementing this algorithm in my code, I particularly found difficulties in trying to create the transpose of the graph in step three. All of the other steps involve simple for-loop traversals and DFS algorithm calls which I did write the code for. I also attempted to utilize BFS instead of DFS since that was already a part of our project. Due to these constraints, I was unable to successfully implement an accurate isStronglyConnected method into my project, following Kosaraju’s Algorithm.

Since this algorithm utilizes depth-first-search, and in my personal attempt I tried to use BFS, both of which have a time complexity of O(|V| + |E|), we know that this sets the upper bound complexity for the overall algorithm. The only remaining elements to check for are the four for-loops that manipulate the “visited” array. These loops all run from i = 0 to i < V, which is of lesser time complexity than the DFS search. After accounting for all of the individual elements of the isStronglyConnected algorithm, we can conclude that the runtime complexity of this algorithm (as implemented by Kosaraju) also results in a **O(|V| + |E|)** time complexity.

My method is provided here, however I have added comments to it that are not within my online submission so as to explain which steps/parts of the algorithm I was able to implement and which ones I was not. Although I had decided to call my BFS method instead of implementing a DFS one, I simply did not include the calls since I was unable to implement step 3 anyway. So this code only shows the four for-loops utilized in steps 1, 2, 4, and 5 of the algorithm to manipulate the “visited” array.

**public** **boolean** isStronglyConnected() {

//Step 1: Initialize vertices as not-visited

Boolean visited[] = **new** Boolean[nVertices];

**for** (**int** i = 0; i < nVertices; i++)

visited[i] = **false**;

//Step 2: Call DFS method here (not implemented)

//Checking if all vertices were visited

**for** (**int** i = 0; i < nVertices; i++)

**if** (visited[i] == **false**)

**return** **false**;

//Step 3: Find Transpose of Graph (not implemented)

//Step 4: Initialize vertices as not-visited

**for** (**int** i = 0; i < nVertices; i++)

visited[i] = **false**;

//Step 5: Call DFS method here (not implemented)

//Checking if all vertices were visited

**for** (**int** i = 0; i < nVertices; i++)

**if** (visited[i] == **false**)

**return** **false**;

//This would normally return true had I been able to implement everything.

**return** **false**;

}

**inputA.txt:**

5

0 1 1

1 0 2

2 0 7

3 2 6

3 4 4

1 3 5

4 1 3

0 2 1

2 4 3

**inputB.txt:**

5

0 1 2

2 0 3

1 2 -1

2 3 1

0 4 4

4 3 -3

3 1 2

**inputC.txt:**

6

3 0 1

3 4 1

4 0 1

4 5 4

0 1 1

0 5 6

5 2 1

1 2 1

2 1 2

2 4 1

**inputD.txt:**

6

2 4 1

4 0 -3

3 5 2

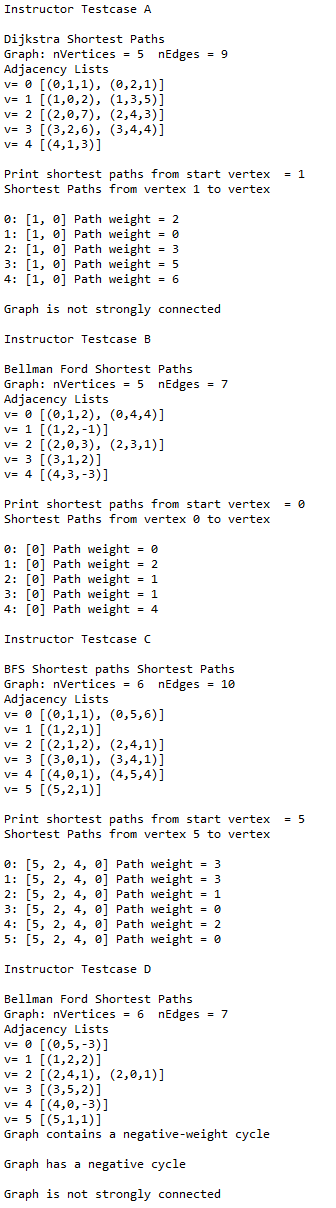
5 1 1

0 5 -3

2 0 1

1 2 2

**Output:**

****

**Graph.java:**

//COMP 482 Project #3

//By Muhammad Ansari

**import** java.util.\*;

**import** java.io.\*;

**public** **class** Graph {

//------------------------------------------------------

**private** ArrayList<EdgeNode>[] adjList;

**private** **int** nVertices;

**private** **int** nEdges;

**private** String fileName;

Scanner f;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructor \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** Graph(String filePath) {

fileName = filePath;

//Scan for file

**try** {

File readFile = **new** File(fileName);

f = **new** Scanner(readFile);

}

**catch**(FileNotFoundException e) {

System.***out***.println("ERR 404: File not found");

}

nVertices = f.nextInt(); //Read number of Vertices

adjList = **new** ArrayList[nVertices]; //Initialize adjacency list

**for**(**int** i = 0; i < adjList.length; i++)

adjList[i] = **new** ArrayList<EdgeNode>(); //Initialize edges in adjacency list

nEdges = 0;

//Read remaining file

**while**(f.hasNextInt()) {

**int** parentVertex = f.nextInt(); //Read parent vertex

**int** childVertex = f.nextInt(); //Read child vertex

**int** weight = f.nextInt(); //Read weight

adjList[parentVertex].add(**new** EdgeNode(parentVertex, childVertex, weight)); //Write into adjacency list

nEdges++; //Increment number of Edges

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Print graph method \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **void** printGraph() {

System.***out***.print("Graph: nVertices = " + nVertices + " " + "nEdges = " + nEdges + "\nAdjacency Lists\n");

//Traverse Adjacency List

**for** (**int** i = 0; i < adjList.length; i++) {

System.***out***.print("v= " + i + " [");

//Traverse and Print Each Edge

**for** (**int** j = 0; j < adjList[i].size(); j++) {

**if**(j != adjList[i].size() -1)

System.***out***.print("(" + adjList[i].get(j).vertex1 + "," + adjList[i].get(j).vertex2 + "," + adjList[i].get(j).weight + "), ");

**else**

System.***out***.print("(" + adjList[i].get(j).vertex1 + "," + adjList[i].get(j).vertex2 + "," + adjList[i].get(j).weight + ")]\n");

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* BFS Shortest paths \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** SPPacket bfsShortestPaths(**int** start) {

//Array Declaration

**boolean**[] visitedArray = **new** **boolean**[nVertices];

**int**[] distanceArray = **new** **int**[nVertices];

**int**[] parentArray = **new** **int**[nVertices];

ArrayList<ArrayList<Integer>> nodeQueue = **new** ArrayList<>(); //Declare queue stack

nodeQueue.add(**new** ArrayList<Integer>()); //Initialize queue

nodeQueue.get(0).add(**new** Integer(start)); //Add starting vertex to queue

**int** i = 0;

//While queue is not empty...

**while** (!nodeQueue.get(i).isEmpty()) {

nodeQueue.add(**new** ArrayList<Integer>());

//For each vertex in the queue...

**for** (Integer j : nodeQueue.get(i)) {

//For each edge of current vertex...

**for** (EdgeNode edges : adjList[j]) {

**int** destinationVertex = edges.vertex2;

//If destination vertex has not yet been visited...

**if** (!visitedArray[destinationVertex]) {

visitedArray[destinationVertex] = **true**; //..visit it

distanceArray[destinationVertex] = distanceArray[j] + edges.weight; //..update distance

parentArray[destinationVertex] = j; //..update parent

nodeQueue.get(i + 1).add(**new** Integer(destinationVertex)); //..add it into queue

}

}

}

i++;

}

//Setting distance as 0 and parent as -1 (NULL) of starting vertex

distanceArray[start] = 0;

parentArray[start] = -1;

SPPacket result = **new** SPPacket(start, distanceArray, parentArray);

**return** result;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Dijkstra's Shortest Path Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** SPPacket dijkstraShortestPaths(**int** start) {

//Array Declaration

**boolean**[] visitedArray = **new** **boolean**[nVertices];

**int**[] distanceArray = **new** **int**[nVertices];

**int**[] parentArray = **new** **int**[nVertices];

//Initialize Arrays

**for** (**int** i = 0; i < nVertices; i++)

{

visitedArray[i] = **false**; //No nodes visited yet, initialized as false

distanceArray[i] = Integer.***MAX\_VALUE***; //All vertex distances marked as infinite

parentArray[i] = -1; //All parent vertices marked as NULL

}

distanceArray[start] = 0; //Setting distance of starting vertex as 0

//Traverse Graph

**for** (**int** i = 0; i < nVertices-1; i++) {

**int** minWeight = Integer.***MAX\_VALUE***;

**int** currentVertex = -1;

//Traverse Graph

**for** (**int** j = 0; j < nVertices; j++)

//If vertex has not been visited and if its distance is less than it's minimum weight...

**if** (visitedArray[j] == **false** && distanceArray[j] <= minWeight) {

minWeight = distanceArray[j]; //update vertex's minimum weight value

currentVertex = j; //record the vertex's index number

}

visitedArray[currentVertex] = **true**; //mark as visited

//Traverse Edges of current Vertex

**for** (**int** j = 0; j < adjList[currentVertex].size(); j++) {

**int** destinationVertex = adjList[currentVertex].get(j).vertex2;

**int** edgeWeight = adjList[currentVertex].get(j).weight;

//If destination vertex has not yet been visited and its distance is greater than current weight...

**if** (!visitedArray[destinationVertex]

&& distanceArray[currentVertex] != Integer.***MAX\_VALUE***

&& distanceArray[currentVertex] + edgeWeight < distanceArray[destinationVertex]) {

distanceArray[destinationVertex] = distanceArray[currentVertex] + edgeWeight; //..update distance

parentArray[destinationVertex] = currentVertex; //..update parent

}

}

}

SPPacket result = **new** SPPacket(start, distanceArray, parentArray);

**return** result;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Bellman Ford Shortest Paths \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** SPPacket bellmanFordShortestPaths(**int** start) {

//Array Declaration

**int**[] distanceArray = **new** **int**[nVertices];

**int**[] parentArray = **new** **int**[nVertices];

//Initialize Arrays

**for** (**int** i = 0; i < nVertices; i++) {

distanceArray[i] = Integer.***MAX\_VALUE***; //All vertex distances marked as infinite

parentArray[i] = -1; //All parent vertices marked as NULL

}

distanceArray[start] = 0; //Setting distance of starting vertex as 0

//Traverse Graph

**for** (**int** i = 0; i < nVertices; i++) {

//Traverse Edges of current Vertex

**for** (**int** j = 0; j < adjList[i].size(); j++) {

**int** currentVertex = adjList[i].get(j).vertex1;

**int** destinationVertex = adjList[i].get(j).vertex2;

**int** edgeWeight = adjList[i].get(j).weight;

//If destination vertex's distance is greater than current weight...

**if** (distanceArray[currentVertex] != Integer.***MAX\_VALUE***

&& distanceArray[currentVertex] + edgeWeight < distanceArray[destinationVertex]) {

distanceArray[destinationVertex] = distanceArray[currentVertex] + edgeWeight; //..update distance

parentArray[destinationVertex] = currentVertex; //..update parent

}

}

}

//Traverse Graph

**for** (**int** i = 0; i < nVertices; i++){

//Traverse Edges of current Vertex

**for** (**int** j = 0; j < adjList[i].size(); j++) {

**int** currentVertex = adjList[i].get(j).vertex1;

**int** destinationVertex = adjList[i].get(j).vertex2;

**int** edgeWeight = adjList[i].get(j).weight;

//If smaller distances still exist, a negative-weight cycle exists. End Graph Algorithm.

**if** (distanceArray[currentVertex] != Integer.***MAX\_VALUE*** &&

distanceArray[currentVertex] + edgeWeight < distanceArray[destinationVertex]) {

System.***out***.println("Graph contains a negative-weight cycle");

**return** **null**;

}

}

}

SPPacket result = **new** SPPacket(start, distanceArray, parentArray);

**return** result;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Prints shortest paths\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **void** printShortestPaths(SPPacket spp){

System.***out***.println("Shortest Paths from vertex " + spp.source + " to vertex\n");

//Traverse Graph

**for**(**int** i = 0; i < nVertices; i++) {

**int**[] shortestPathArray = **new** **int**[nEdges];

**int** currentVertex = 0;

**int** k = 0;

//While currentVertex is not the source Vertex

**while**(currentVertex != -1) {

shortestPathArray[k] = currentVertex;

currentVertex = spp.parent[currentVertex];

k++;

}

System.***out***.print(i + ": [");

//Traverse Shortest Path backwards and print

**for**(**int** j = k - 1; j >= 0; j--) {

**if** (j != 0)

System.***out***.print(shortestPathArray[j] + ", ");

**else**

System.***out***.print(shortestPathArray[j] + "] Path weight = " + spp.d[i] + "\n");

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*isStronglyConnected\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **boolean** isStronglyConnected() {

Boolean visited[] = **new** Boolean[nVertices];

**for** (**int** i = 0; i < nVertices; i++)

visited[i] = **false**;

**for** (**int** i = 0; i < nVertices; i++)

**if** (visited[i] == **false**)

**return** **false**;

**for** (**int** i = 0; i < nVertices; i++)

visited[i] = **false**;

**for** (**int** i = 0; i < nVertices; i++)

**if** (visited[i] == **false**)

**return** **false**;

**return** **false**;

}//end Graph class

//place the EdgeNode class and the SPPacket class inside the Graph.java file

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**class** EdgeNode {

**int** vertex1;

**int** vertex2;

**int** weight;

**public** EdgeNode(**int** v1, **int** v2, **int** w) {

vertex1 = v1;

vertex2 = v2;

weight = w;

}

@Override

**public** String toString() {

String edgeInfo = "Parent Vertex: " + vertex1 + "\nChild Vertex: " + vertex2 + "\nWeight: " + weight;

**return** edgeInfo;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**class** SPPacket {

**int**[] d; //distance array

**int**[] parent; //parent path array

**int** source; //source vertex

**public** SPPacket(**int** start, **int**[] distance, **int**[] pp) {

source = start;

d = distance;

parent = pp;

}

**public** **int**[] getDistance() {

**return** d;

}

**public** **int**[] getParent() {

**return** parent;

}

**public** **int** getSource() {

**return** source;

}

**public** String toString() {

String packetInfo = "Source Vertex: " + source + "\nDistance Array: [";

**for**(**int** i = 0; i < d.length; i++){

**if** (i != d.length - 1)

packetInfo += d[i] + ", ";

**else**

packetInfo += d[i] + "]\nParent Array: [ ";

}

**for**(**int** i = 0; i < parent.length; i++){

**if**(i != parent.length -1)

packetInfo += parent[i] + ", ";

**else**

packetInfo += parent[i] + "]";

}

**return** packetInfo;

}

}

**public** **static** **void** main(String...args){

System.***out***.println("Instructor Testcase A");

System.***out***.println("\nDijkstra Shortest Paths");

Graph g1 = **new** Graph("C:\\Users\\Faiz\\Desktop\\inputA.txt");

g1.printGraph();

**int** start1 = 1;

SPPacket spp1 = g1.dijkstraShortestPaths(start1);

System.***out***.println("\nPrint shortest paths from start vertex = " + start1);

g1.printShortestPaths( spp1 );

**if**( g1.isStronglyConnected())

System.***out***.println( "\nGraph is strongly connected");

**else**

System.***out***.println( "\nGraph is not strongly connected");

System.***out***.println("Instructor Testcase B");

System.***out***.println("\nBellman Ford Shortest Paths");

Graph g2 = **new** Graph("C:\\Users\\Faiz\\Desktop\\inputB.txt");

g2.printGraph();

**int** start2 = 0;

SPPacket spp2 = g2.bellmanFordShortestPaths(start2);

**if**( spp2 != **null**)

{

System.***out***.println("\nPrint shortest paths from start vertex = " + start2);

g2.printShortestPaths( spp2 );

}

**else**

System.***out***.println("Graph has a negative cycle");

System.***out***.println("Instructor Testcase C");

System.***out***.println("\nBFS Shortest paths Shortest Paths");

Graph g3 = **new** Graph("C:\\Users\\Faiz\\Desktop\\inputC.txt");

g3.printGraph();

**int** start3 = 5;

SPPacket spp3 = g3.bfsShortestPaths(start3);

System.***out***.println("\nPrint shortest paths from start vertex = " + start3);

g3.printShortestPaths( spp3 );

System.***out***.println("Instructor Testcase D");

System.***out***.println("\nBellman Ford Shortest Paths");

Graph g4 = **new** Graph("C:\\Users\\Faiz\\Desktop\\inputD.txt");

g4.printGraph();

**int** start4 = 0;

SPPacket spp4 = g4.bellmanFordShortestPaths(start4);

**if**( spp4 != **null**)

{

System.***out***.println("\nPrint shortest paths from start vertex = " + start4);

g1.printShortestPaths( spp4 );

}

**else**

System.***out***.println("\nGraph has a negative cycle");

**if**( g4.isStronglyConnected())

System.***out***.println( "\nGraph is strongly connected");

**else**

System.***out***.println( "\nGraph is not strongly connected");

}

}