**Algorithms Assignment 3**

**Group 6**

Will Hauber

Matthew McClellan

John Mikolay

11/5/2016

**Source**

/\*

\* Team Members:

\* Will Hauber

\* Matthew McClellan

\* John Mikolay

\*

\* Date: Nov 6, 2016

\*

\* Assignment: 3

\*/

#include <iostream>

#include <queue>

#include <stdlib.h>

#include <string>

#include <vector>

using namespace std;

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

// Graph Structs //

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

struct graphEdge

{

int firstNode;

int secondNode;

};

struct graphMatrix

{

int numberOfNodes;

vector< vector<int>\* >\* matrix;

};

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

// Graph Method Signatures //

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

graphMatrix generateAdjacencyMatrix(int numNodes, vector<graphEdge> edges);

graphMatrix generateDistanceMatrix(graphMatrix adjacencyMatrix);

vector<int> getNeighbors(int node, graphMatrix graphAdjacencyMatrix);

vector<int> getDistancesUsingBfs(int node, graphMatrix graphAdjacencyMatrix);

int getPathDistanceUsingBfs(int node1, int node2, graphMatrix graphAdjacencyMatrix);

vector< vector<int>\* >\* getConnectedComponents (graphMatrix distanceMatrix);

int graphDiameter(graphMatrix distanceMatrix);

void printMatrix(graphMatrix matrix);

void deleteMatrix(graphMatrix matrix);

void deleteMatrix(vector< vector<int>\* >\* matrix);

//------//

// Main //

//------//

int main(int argc, char\* argv[])

{

// Parse command line arguments

string usageString = "Usage: ./a.out <numNodes> <edge1Node1> <edge1Node2> <edge2Node1 <edge2Node2> ... -1";

if (argc <= 1)

{

cout << usageString << endl;

return 1;

}

int numberOfNodes = atoi(argv[1]);

cout << "Number of nodes: " << numberOfNodes << endl;

vector<graphEdge> edges;

for (int ii = 2 /\* skip to the edges args \*/; ii < argc; ii += 2)

{

// See if we're at the end

if (atoi(argv[ii]) < 0)

{

break;

}

// Make sure there are at least two more items

if (ii + 1 == argc)

{

cout << usageString << endl;

return 2;

}

// Make sure the next item isn't the end

if (atoi(argv[ii+1]) < 0)

{

cout << usageString << endl;

return 3;

}

//Error checking: Make sure node is not larger than the total number of nodes

if (atoi(argv[ii]) >= numberOfNodes)

{

cout << "Input Error: Node number " << atoi(argv[ii]) << " is too large. Valid node numbers are 0-" << numberOfNodes - 1 << "." << endl;

return 0;

}

else if(atoi(argv[ii+1]) >= numberOfNodes)

{

cout << "Input Error: Node number " << atoi(argv[ii+1]) << " is too large. Valid node numbers are 0-" << numberOfNodes - 1 << "." << endl;

return 0;

}

// Otherwise, add the edge

graphEdge newEdge;

newEdge.firstNode = atoi(argv[ii]);

newEdge.secondNode = atoi(argv[ii+1]);

edges.push\_back(newEdge);

}

for (int ii = 0; ii < edges.size(); ii++)

{

cout << "Edge " << ii << ": " << edges[ii].firstNode << " " << edges[ii].secondNode << endl;

}

cout << endl;

graphMatrix adjacencyMatrix = generateAdjacencyMatrix(numberOfNodes, edges);

cout << "Adjacency Matrix:" << endl;

printMatrix(adjacencyMatrix);

cout << endl;

graphMatrix distanceMatrix = generateDistanceMatrix(adjacencyMatrix);

cout << "Distance Matrix:" << endl;

printMatrix(distanceMatrix);

cout << endl;

cout << "Graph Diameter: " << graphDiameter(distanceMatrix) << endl;

cout << endl;

vector< vector<int>\* >\* connectedComponents = getConnectedComponents(distanceMatrix);

deleteMatrix(adjacencyMatrix);

deleteMatrix(distanceMatrix);

deleteMatrix(connectedComponents);

return 0;

}

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

// Graph Method Definitions //

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

graphMatrix generateAdjacencyMatrix(int numNodes, vector<graphEdge> edges)

{

graphMatrix adjacencyMatrix;

adjacencyMatrix.numberOfNodes = numNodes;

adjacencyMatrix.matrix = new vector< vector<int>\* >();

// Build the matrix row for each node

for (int currentNode = 0; currentNode < numNodes; currentNode++)

{

vector<int>\* matrixRow = new vector<int>();

// Initialize the row

for (int ii = 0; ii < numNodes; ii++)

{

matrixRow->push\_back(0);

}

// Find adjancent nodes

for (int edgeIndex = 0; edgeIndex < edges.size(); edgeIndex++)

{

// If the current node participates in this edge, it is adjacent to the other node on the edge

graphEdge currentEdge = edges[edgeIndex];

if (currentEdge.firstNode == currentNode)

{

(\*matrixRow)[currentEdge.secondNode] = 1;

}

if (currentEdge.secondNode == currentNode)

{

(\*matrixRow)[currentEdge.firstNode] = 1;

}

}

// Add the row to the matrix

adjacencyMatrix.matrix->push\_back(matrixRow);

}

return adjacencyMatrix;

}

vector<int> getNeighbors(int node, graphMatrix graphAdjacencyMatrix)

{

vector<int>\* matrixRowForNode = (\*graphAdjacencyMatrix.matrix)[node];

vector<int> neighbors;

for (int ii = 0; ii < matrixRowForNode->size(); ii++)

{

if ((\*matrixRowForNode)[ii] == 1)

{

neighbors.push\_back(ii);

}

}

return neighbors;

}

vector<int> getDistancesUsingBfs(int node, graphMatrix graphAdjacencyMatrix)

{

// Initialize the distances vector

vector<int> distances;

for (int ii = 0; ii < graphAdjacencyMatrix.numberOfNodes; ii++)

{

// Assume we can't reach each node to start with

distances.push\_back(-1);

}

// We're distance zero from ourself

distances[node] = 0;

// Initialize the open list

queue<int> openList;

openList.push(node);

// Start searching

while (openList.size() > 0)

{

// Remove the next node in the queue

int currentNode = openList.front();

openList.pop();

int currentDistance = distances[currentNode];

// Loop throught this node's neighbors

vector<int> neighbors = getNeighbors(currentNode, graphAdjacencyMatrix);

for (int ii = 0; ii < neighbors.size(); ii++)

{

int neighbor = neighbors[ii];

// Skip this neighbor if we've already visited it

if (distances[neighbor] != -1)

{

continue;

}

// Set the distance this neighbor, then add it to the open list

distances[neighbor] = currentDistance + 1;

openList.push(neighbor);

}

}

return distances;

}

int getPathDistanceUsingBfs(int node1, int node2, graphMatrix graphAdjacencyMatrix)

{

vector<int> node1Distances = getDistancesUsingBfs(node1, graphAdjacencyMatrix);

return node1Distances[node2];

}

void printMatrix(graphMatrix matrixToPrint)

{

for (int ii = 0; ii < matrixToPrint.matrix->size(); ii++)

{

vector<int>\* currentRow = (\*matrixToPrint.matrix)[ii];

for (int jj = 0; jj < currentRow->size(); jj++)

{

//Print numbers, spaces included to keep columns aligned

if((\*currentRow)[jj] == -1)

{

cout << (\*currentRow)[jj] << " ";

}

else

{

cout << " " << (\*currentRow)[jj] << " ";

}

}

cout << endl;

}

}

void deleteMatrix(graphMatrix matrixToDelete)

{

for (int ii = 0; ii < matrixToDelete.matrix->size(); ii++)

{

delete (\*matrixToDelete.matrix)[ii];

}

delete matrixToDelete.matrix;

}

void deleteMatrix(vector< vector<int>\* >\* matrixToDelete)

{

for (int ii = 0; ii < matrixToDelete->size(); ii++)

{

delete matrixToDelete->at(ii);

}

delete matrixToDelete;

}

graphMatrix generateDistanceMatrix(graphMatrix adjacencyMatrix)

{

graphMatrix distanceMatrix;

distanceMatrix.numberOfNodes = adjacencyMatrix.numberOfNodes;

distanceMatrix.matrix = new vector< vector<int>\* >();

for (int row = 0 ; row < adjacencyMatrix.numberOfNodes ; row++)

{

//getDistancesUsingBfs makes each row of the distanceMatrix for us. just save return

vector<int> tempCurrentDistances = getDistancesUsingBfs(row, adjacencyMatrix);

vector<int>\* currentDistances = new vector<int>();

//deep copy tempCurrentDistances into currentDistances

for (int ii = 0; ii < tempCurrentDistances.size() ; ii++)

{

currentDistances->push\_back( tempCurrentDistances.at(ii) );

}

//add currentDistances to dstanceMatrix

distanceMatrix.matrix->push\_back(currentDistances);

}

return distanceMatrix;

}

vector< vector<int>\* >\* getConnectedComponents(graphMatrix distanceMatrix)

{

vector< vector<int>\* >\* connectedComponents = new vector< vector<int>\* >();

vector<bool>\* visited = new vector<bool>(distanceMatrix.matrix ->size(), 0);

for (int row = 0; row < distanceMatrix.matrix ->size(); row++)

{

//if this node has never been explored

if (visited->at(row) == 0)

{

//set node as explored

(\*visited)[row] = 1;

//make component vector to keep track of this node and nodes it is connected to

vector<int>\* currentComponent = new vector<int>();

currentComponent->push\_back(row);

//add this component to vector of components

connectedComponents->push\_back(currentComponent);

vector<int>\* currentRow = distanceMatrix.matrix ->at(row);

for(int column = 0; column < currentRow->size(); column++)

{

//if the row node is connected to the node referenced by this column,

// add the column node to the connected component vector. Set the column

// node as visited.

if( currentRow->at(column) > -1 && column != row)

{

currentComponent->push\_back(column);

(\*visited)[column] = 1;

}

}

}

}

//print out components

cout << "ConnectedComponentMatrix:" << endl;

for (int ii = 0; ii < connectedComponents->size(); ii++)

{

vector<int>\* currentRow = connectedComponents->at(ii);

for (int jj = 0; jj < currentRow->size(); jj++)

{

cout << currentRow->at(jj) << " ";

}

cout << endl;

}

return connectedComponents;

}

int graphDiameter(graphMatrix distanceMatrix)

{

int max = 0;

//iterate through all the distance pairs and find the largest value.

for(int row = 0; row < distanceMatrix.matrix->size(); row++)

{

for (int column = 0; column < distanceMatrix.matrix->at(row)->size(); column++)

{

int currentDistance = distanceMatrix.matrix->at(row)->at(column);

if(currentDistance > max)

{

max = currentDistance;

}

else if(currentDistance == -1)

{

// If graph is not connected return -1

return currentDistance;

}

}

}

return max;

}

**Connected Graph**

Input:

10 0 1 1 2 2 3 0 4 4 5 0 6 6 7 7 8 0 9 -1

Output:

Number of nodes: 10

Edge 0: 0 1

Edge 1: 1 2

Edge 2: 2 3

Edge 3: 0 4

Edge 4: 4 5

Edge 5: 0 6

Edge 6: 6 7

Edge 7: 7 8

Edge 8: 0 9

Adjacency Matrix:

0 1 0 0 1 0 1 0 0 1

1 0 1 0 0 0 0 0 0 0

0 1 0 1 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0

1 0 0 0 0 1 0 0 0 0

0 0 0 0 1 0 0 0 0 0

1 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 1 0 1 0

0 0 0 0 0 0 0 1 0 0

1 0 0 0 0 0 0 0 0 0

Distance Matrix:

0 1 2 3 1 2 1 2 3 1

1 0 1 2 2 3 2 3 4 2

2 1 0 1 3 4 3 4 5 3

3 2 1 0 4 5 4 5 6 4

1 2 3 4 0 1 2 3 4 2

2 3 4 5 1 0 3 4 5 3

1 2 3 4 2 3 0 1 2 2

2 3 4 5 3 4 1 0 1 3

3 4 5 6 4 5 2 1 0 4

1 2 3 4 2 3 2 3 4 0

Graph Diameter: 6

Connected Components:

0 1 2 3 4 5 6 7 8 9

**Disconnected Graph**

Input:

10 0 1 1 2 2 3 4 5 4 6 4 7 8 9 -1

Output:

Number of nodes: 10

Edge 0: 0 1

Edge 1: 1 2

Edge 2: 2 3

Edge 3: 4 5

Edge 4: 4 6

Edge 5: 4 7

Edge 6: 8 9

Adjacency Matrix:

0 1 0 0 0 0 0 0 0 0

1 0 1 0 0 0 0 0 0 0

0 1 0 1 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 1 1 1 0 0

0 0 0 0 1 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 1 0

Distance Matrix:

0 1 2 3 -1 -1 -1 -1 -1 -1

1 0 1 2 -1 -1 -1 -1 -1 -1

2 1 0 1 -1 -1 -1 -1 -1 -1

3 2 1 0 -1 -1 -1 -1 -1 -1

-1 -1 -1 -1 0 1 1 1 -1 -1

-1 -1 -1 -1 1 0 2 2 -1 -1

-1 -1 -1 -1 1 2 0 2 -1 -1

-1 -1 -1 -1 1 2 2 0 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1 0 1

-1 -1 -1 -1 -1 -1 -1 -1 1 0

Graph Diameter: -1

Connected Components:

0 1 2 3

4 5 6 7

8 9