using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.IO;

namespace Bingo

{

class Program

{

private static RelationshipGraph rg;

// Read RelationshipGraph whose filename is passed in as a parameter.

// Build a RelationshipGraph in RelationshipGraph rg

private static void ReadRelationshipGraph(string filename)

{

rg = new RelationshipGraph(); // create a new RelationshipGraph object

string name = ""; // name of person currently being read

int numPeople = 0;

string[] values;

Console.Write("Reading file " + filename + "\n");

try

{

string input = System.IO.File.ReadAllText(filename);// read file

input = input.Replace("\r", ";"); // get rid of nasty carriage returns

input = input.Replace("\n", ";"); // get rid of nasty new lines

string[] inputItems = Regex.Split(input, @";\s\*"); // parse out the relationships (separated by ;)

foreach (string item in inputItems)

{

if (item.Length > 2) // don't bother with empty relationships

{

values = Regex.Split(item, @"\s\*:\s\*"); // parse out relationship:name

if (values[0] == "name") // name:[personname] indicates start of new person

{

name = values[1]; // remember name for future relationships

rg.AddNode(name); // create the node

numPeople++;

}

else

{

rg.AddEdge(name, values[1], values[0]); // add relationship (name1, name2, relationship)

// handle symmetric relationships -- add the other way

if (values[0] == "spouse" || values[0] == "hasFriend")

rg.AddEdge(values[1], name, values[0]);

// for parent relationships add child as well

else if (values[0] == "parent")

rg.AddEdge(values[1], name, "child");

else if (values[0] == "child")

rg.AddEdge(values[1], name, "parent");

}

}

}

}

catch (Exception e)

{

Console.Write("Unable to read file {0}: {1}\n", filename, e.ToString());

}

Console.WriteLine(numPeople + " people read");

}

// Show the relationships a person is involved in

private static void ShowPerson(string name)

{

GraphNode n = rg.GetNode(name);

if (n != null)

Console.Write(n.ToString());

else

Console.WriteLine("{0} not found", name);

}

// Show a person's friends

private static void ShowFriends(string name)

{

GraphNode n = rg.GetNode(name);

if (n != null)

{

Console.Write("{0}'s friends: ",name);

List<GraphEdge> friendEdges = n.GetEdges("hasFriend");

foreach (GraphEdge e in friendEdges) {

Console.Write("{0} ",e.To());

}

Console.WriteLine();

}

else

Console.WriteLine("{0} not found", name);

}

// Show the orphans of a Relationship Graph

private static void ShowOrphans()

{

int numOrphans = 0; //Create a counter variable to count the number of orphans in a Relationship Graph

List<GraphNode> orphanList = new List<GraphNode>(); //Create a new list to store the Orphans in

foreach (GraphNode node in rg.nodes) { //Loop through the list of nodes in the Relationship Graph

List<GraphEdge> parentEdges = node.GetEdges("hasParent");

if (parentEdges.Count == 0) { //Check to see if the size of the hasParent edge List is zero (i.e. meaning they have no parents)

orphanList.Add(node); //Add the node to the Orphan List

numOrphans++; //Increment in the orphan counter

}

}

string numberOrphans = Convert.ToString(numOrphans);

foreach (GraphNode node in orphanList) //For each orphan node in the List, print out the name of the orphan

{

Console.Write(node.Name() + '\n');

}

Console.Write("There are " + numberOrphans + " orphans in the GraphRelationship."); //Inform the user how many orphans were found

}

//List the descendants of an indvidual

private static void ListDescendants(string name)

{

GraphNode node = rg.GetNode(name); //Get the node from the Relationship Graph of the name of the individual

int generation\_count = 0; //Set the generation count to be zero

//If the individual that the user is searching for does not exist in the current Relationship Graph

if (node == null)

{

Console.Write("{0} not found\n", name);

}

else if (node.GetEdges("hasChild").Count == 0) //If the node has no hasChild edge, print that there are no descendants

{

Console.Write("{0} has no descendants\n", name);

}

//Else, list out the descendants of the individual

else

{

List<GraphEdge> descendantsList = node.GetEdges("hasChild"); //Get the edges hasChild of the individual

List<GraphNode> nextDescendants = new List<GraphNode>(); //Create two new lists of GraphNodes, one for the current generation of

List<GraphNode> currentDescendants = new List<GraphNode>(); //descendants, and one for the following generation of descendants

foreach (GraphEdge edge in descendantsList) //For every edge hasChild, add the node that the edge is directed towards

{ //to the current descendants list of individuals

currentDescendants.Add( rg.GetNode( edge.To() ) );

}

while (currentDescendants.Count > 0) //As long as the currentDescendants List is not zero

{

//The following if, if-else, and else statements determine the label of descendants to write to the console

if (generation\_count == 0)

{

Console.Write(node.Name() + " Children:\n");

}

else if (generation\_count == 1)

{

Console.Write(node.Name() + " Grandchildren:\n");

}

else

{

Console.Write(node.Name());

for (int i = 0; i < generation\_count; i++)

{

Console.Write(" Great");

}

Console.Write(" GrandChildren:\n");

}

//For each node in the current descendants list

foreach (GraphNode descendant in currentDescendants)

{

Console.Write("{0}\n", descendant.Name()); //Write out the name of the descendant to the console

descendantsList = descendant.GetEdges("hasChild"); //Get a list of the edges hasChild from that descendant

foreach (GraphEdge edge1 in descendantsList) //For each of those edges hasChild of the current descendant

{

nextDescendants.Add( rg.GetNode( edge1.To() ) ); //Add the node that the edge hasChild is directed towards

}

}

currentDescendants = nextDescendants; //Copy the nextDescendant List nodes to the currentDescandents List

nextDescendants = new List<GraphNode>(); //Clear our the nextDescendant List

generation\_count++; //Increment the generation counter to be used for future labeling of descendants

}

}

}

// Find the shortest path of relationships between two people

private static void Bingo(string start\_person, string end\_person)

{

//If the start and end person are the same person, inform the user

if (start\_person == end\_person)

{

Console.Write(start\_person + " and " + end\_person + " are the same person.\n");

}

else

{

int depth\_counter = 0; //Variable to keep track the minimum depth to get to end\_person from start\_person

Dictionary<string, Boolean> dictionary = new Dictionary<string, Boolean>(); //new Dictionary to see if a node has been visited before

foreach (GraphNode node\_ in rg.nodes)

dictionary.Add(node\_.Name(), false);

Queue<string> q = new Queue<string>(); //Create a new queue

dictionary[start\_person] = true; //Mark start\_person as visited

q.Enqueue(start\_person); //Add the start\_person to the queue

//As long as the queue is not empty

while (q.Count > 0)

{

string name\_one = q.Dequeue(); //Remove the next element in the queue

GraphNode node\_one = rg.GetNode(name\_one);

List<GraphEdge> AdjacentEdges = node\_one.GetEdges();

depth\_counter++; //Increment the depth counter as we went one depth further

//Check each GraphEdge coming out of the current node

foreach (GraphEdge edge in AdjacentEdges)

{

if (name\_one == end\_person) //If the current node and the end\_person are the same, break from the While Loop

break;

if (dictionary[edge.To()] == false)

{

dictionary[edge.To()] = true; //Update the visited dictionary

q.Enqueue(edge.To()); //Add it to the queue

}

}

}

List<GraphNode> list\_nodes = new List<GraphNode>(); //New lists for the nodes and the edges of the path to the end\_person

List<GraphEdge> list\_edges = new List<GraphEdge>();

Dictionary<string, Boolean> dictionary2 = new Dictionary<string, Boolean>();

foreach (GraphNode node\_ in rg.nodes)

dictionary2.Add(node\_.Name(), false);

int stack\_depth = 0; //Stack depth counter to keep track how far to go in the DFS

Stack<GraphNode> stack\_nodes = new Stack<GraphNode>(); //Create a new stack

stack\_nodes.Push(rg.GetNode(start\_person)); //Push the start\_person onto the stack

while (true)

{

while (stack\_nodes.Count > 0) //As long as the stack is not empty

{

GraphNode node\_one = stack\_nodes.Pop(); //Pop off the next element

list\_nodes.Add(node\_one); //Add it to the list

List<GraphEdge> edges\_list = node\_one.GetEdges();

list\_edges.Add(edges\_list[0]);

foreach (GraphEdge edge\_two in edges\_list)

{

stack\_nodes.Push(rg.GetNode(edge\_two.To())); //Add each node coming off the edges to the Stack

}

stack\_depth++;

if (stack\_depth == depth\_counter) //If the two depth counters are equal, break

break;

}

//Check to see if the first node in the list is the same as the start\_person

// and if the last node in the list is the same as the end\_person

if (list\_nodes[0] == rg.GetNode(start\_person) && list\_nodes[depth\_counter - 1] == rg.GetNode(end\_person))

{

break;

}

}

//Printing out the relationships between the start\_person and the end\_person

int i = 0;

foreach (GraphNode NODE in list\_nodes)

{

GraphEdge edge\_label = list\_edges[i];

Console.Write(NODE.Name() + edge\_label.Label() + list\_nodes[i + 1] + "\n");

i++;

}

}

}

// accept, parse, and execute user commands

private static void commandLoop()

{

string command = "";

string[] commandWords;

Console.Write("Welcome to Harry's Dutch Bingo Parlor!\n");

while (command != "exit")

{

Console.Write("\nEnter a command: ");

command = Console.ReadLine();

commandWords = Regex.Split(command, @"\s+"); // split input into array of words

command = commandWords[0];

if (command == "exit")

; // do nothing

// read a relationship graph from a file

else if (command == "read" && commandWords.Length > 1)

ReadRelationshipGraph(commandWords[1]);

// show information for one person

else if (command == "show" && commandWords.Length > 1)

ShowPerson(commandWords[1]);

// show all the friends of a person

else if (command == "friends" && commandWords.Length > 1)

ShowFriends(commandWords[1]);

// show all the people who are orphans (have no parents)

else if (command == "orphans")

ShowOrphans();

// show all the descendents of an individual

else if (command == "descendants" && commandWords.Length > 1)

ListDescendants(commandWords[1]);

// find the shortest path of relationships between two people

//else if (command == "bingo" && commandWords.Length > 1)

//Bingo(commandWords[1], commandWords[2]);

// dump command prints out the graph

else if (command == "dump")

rg.dump();

// illegal command

else

Console.Write("\nLegal commands: read [filename], dump, show [personname],\n friends [personname], exit\n");

}

}

static void Main(string[] args)

{

commandLoop();

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Bingo

{

/// <summary>

/// Represents a directed labeled graph with a string name at each node

/// and a string label for each edge.

/// </summary>

class RelationshipGraph

{

/\*

\* This data structure contains a list of nodes (each of which has

\* an adjacency list) and a dictionary (hash table) for efficiently

\* finding nodes by name

\*/

/\* nodes property should probably be private, with accessor

\* functions, but it's public at the moment.

\*/

public List<GraphNode> nodes;

private Dictionary<String, GraphNode> nodeDict;

// constructor builds empty relationship graph

public RelationshipGraph()

{

nodes = new List<GraphNode>();

nodeDict = new Dictionary<String,GraphNode>();

}

// AddNode creates and adds a new node if there isn't already one by that name

public void AddNode(string name)

{

if (!nodeDict.ContainsKey(name))

{

GraphNode n = new GraphNode(name);

nodes.Add(n);

nodeDict.Add(name, n);

}

}

// AddEdge adds the edge, creating endpoint nodes if necessary.

// Edge is added to adjacency list of from edges.

public void AddEdge(string name1, string name2, string relationship)

{

AddNode(name1); // create the node if it doesn't already exist

GraphNode n1 = nodeDict[name1]; // now fetch a reference to the node

AddNode(name2);

GraphNode n2 = nodeDict[name2];

GraphEdge e = new GraphEdge(n1, n2, relationship);

n1.AddIncidentEdge(e);

}

// Get a node by name using dictionary

public GraphNode GetNode(string name)

{

if (nodeDict.ContainsKey(name))

return nodeDict[name];

else

return null;

}

// Return a text representation of graph

public void dump()

{

foreach (GraphNode n in nodes)

{

Console.Write(n.ToString());

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Bingo

{

/// <summary>

/// Represents a node in a RelationshipGraph

/// </summary>

class GraphNode

{

private string name;

private List<GraphEdge> incidentEdges;

// constructor

public GraphNode(string v)

{

name = v;

incidentEdges = new List<GraphEdge>();

}

// Add an edge (but don't add duplicate edges)

public void AddIncidentEdge(GraphEdge e)

{

foreach (GraphEdge edge in incidentEdges)

{

if (edge.ToString() == e.ToString())

return;

}

incidentEdges.Add(e);

}

// return a list of all outgoing edges

public List<GraphEdge> GetEdges()

{

return incidentEdges;

}

// return a list of outgoing edges of specified label

public List<GraphEdge> GetEdges(string label)

{

List<GraphEdge> list = new List<GraphEdge>();

foreach (GraphEdge e in incidentEdges)

if (e.Label() == label)

list.Add(e);

return list;

}

// return text form of node, including outgoing edges

public override string ToString()

{

string result = name + "\n";

foreach (GraphEdge e in incidentEdges)

{

result = result + " " + e.ToString() + "\n";

}

return result;

}

// return name of node

public string Name()

{

return name;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Bingo

{

/// <summary>

/// Represents a labeled, directed edge in a RelationshipGraph

/// </summary>

class GraphEdge

{

private string label;

private GraphNode fromNode, toNode;

// constructor

public GraphEdge(GraphNode from, GraphNode to, string myLabel)

{

fromNode = from;

toNode = to;

label = myLabel;

}

// return label of edge

public string Label()

{

return label;

}

// return the name of the "to" person in the relationship

public string To()

{

return toNode.Name();

}

// return string form of edge

public override string ToString()

{

string result = fromNode.Name() + " --(" + label + ")--> " + toNode.Name();

return result;

}

}

}