Computational Analysis of Graphs in C++ to Determine if they are a Tree

Jose Gonzalez*

In graph theory, graphs can be a tree by having various characteristics. A tree is defined as a connected graph that contains no cycles. In the present paper we will go over how to do the computational analysis of graphs in C++. We will review the methods used in the application and give future insight of possible improvements and applications.

I. INTRODUCTION

Tree is a characterization of a graph. This characterization is given to a connected graph that has no cycles. In order to come to the conclusion of whether a graph is a tree we must first evaluate whether it meets these requirements. Only after the evaluation we can confidently come to the conclusion. The complexity of coming to this conclusion may not seem trivial, but the implementation of a system in order to process graphs and algorithms to determine if it's a tree through computational analysis is. We will go over the developed application in order to determine whether a graph is a tree and test the accuracy of the application.

In the present paper, we will also review the system and algorithms used in the application in order to let users input a graph and test whether the graph meets the requirements of being a tree. Specifically, we will test whether a graph is connected and acyclic. Once we have reviewed the algorithms used we will also provide different graphs to the application in order to establish that it works. Once we have established that it works we will review the application and discuss the future applications and improvement of it.

This paper is organized as follows. Section 2 will provide background information on trees and practical methods used in order to determine if a graph is a tree. Section 3 of the paper will showcase the application and how to properly use it. Section 4 goes over the algorithm and the system used in order to process and determine if a graph is a tree. In Section 5 we will conclude whether the application works and its accuracy by testing eight different graphs with each known characteristics and checking whether they match with the output of the application. Finally, section 6 the conclusion, will go over the future improvements and applications of the application.

*Electronic address: `jose.gonzalez39@utrgv.edu`

II. DEFINING A GRAPH AS A TREE

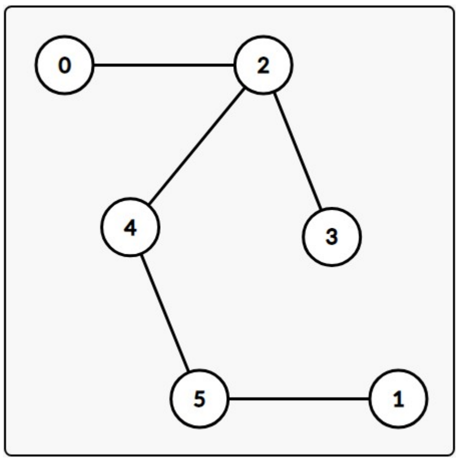


FIG. 1: Acyclic Graph

In graph theory, a tree is a connected graph that contains no cycles such as in FIG. 1. A graph with no cycles can be described as acyclic. In order for a graph to be acyclic it must follow the condition in which any two vertices are connected by exactly one path.

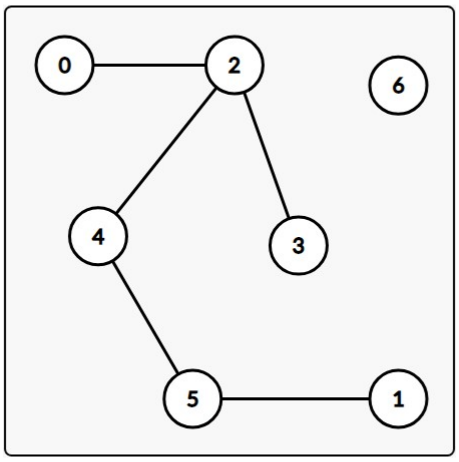


FIG. 2: Disconnected Graph

Once it is determined that the graph is acyclic it must meet the condition of being a connected graph. A disconnected graph is a graph with two components such as in FIG. 2 with vertex 6 being the second component. A graph is said to be connected if it has one component and has one path between every pair of vertices.

A. Initial Conditions of a Tree

Initial conditions to note of a tree that will always remain the same are seen in the number of edges $E(G)$, which are always equal to $V(G)-1$, making $E(G) = V(G)-1$. Although this does not prove that the graph is a tree, it will always be true if it is a tree. Since a graph where any two vertices are connected by exactly one path will always have edges equal to the number of vertices minus one.

B. Method of Determining if a Graph is a Tree

Typically speaking, in graph theory, most characteristics of a graph can be concluded by traversing a graph either manually or systematically through a program. If we for example manually traverse a graph with,
$V(G) = \{a, b, c\}$
$E(G) = \{ab, bc\}$
Start from an initial vertex $\{a\}$ and visit every adjacent vertex of $\{a, b, c\}$ and determine that any two vertices are connected by exactly one path we can conclude that the graph is a tree. If we encounter a vertex more than once then that means that there must be a cycle in the graph disqualifying it from being a tree. If we do not encounter one of the vertex then that means that the graph contains at least one or more vertices that is not connected, therefore the graph is not connected thus disqualifying it from being a tree. If either condition is not met then the graph is not a tree and if both conditions are met then the graph is a tree.

III. APPLICATION AND USE

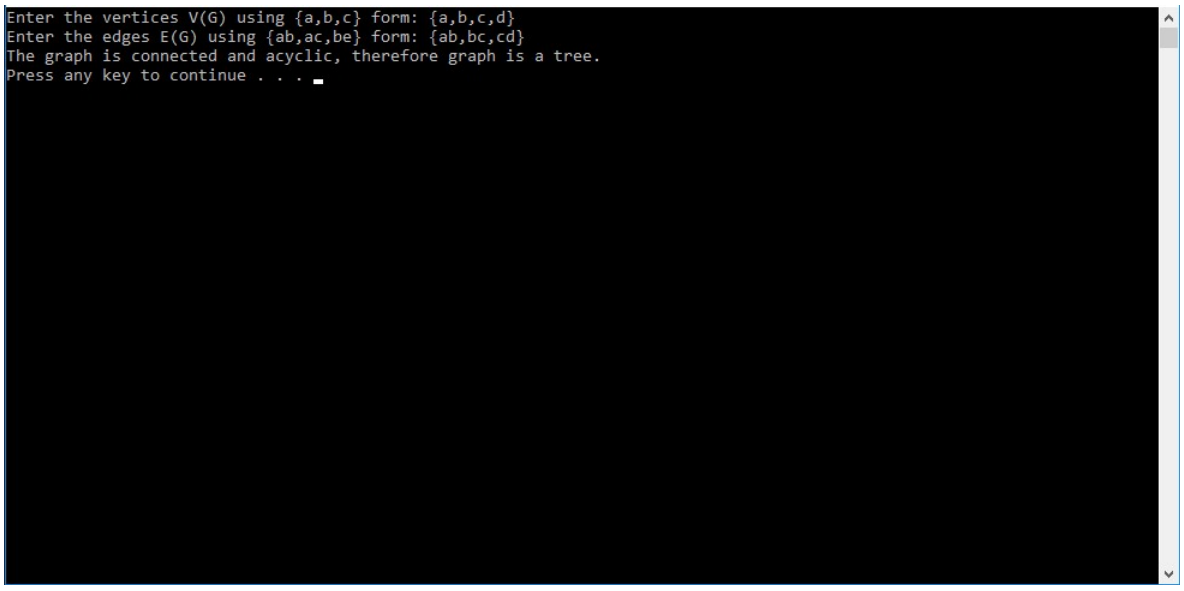

```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d}
Enter the edges E(G) using {ab,ac,be} form: {ab,bc,cd}
The graph is connected and acyclic, therefore graph is a tree.
Press any key to continue . . .
```

FIG. 3: Image of Program

The application was programmed in C++ using Visual Studio 2017. A criteria of the program was to ensure that it could take the users input allowing distinct graphs to be processed. Although increasing the complexity of the program the outcome was exceptional allowing the user to input any graph using the $V(G) = \{a, b, c\}$ notation for vertices and $E(G) = \{ab, bc\}$ notation for edges. Once the user has inputted the vertices and edges of the graph, the program will run an algorithm recursively checking the vertices and determining if it contains cycles and is connected or disconnected. Once it has finished processing the graph, it will state whether it is connected or disconnected and whether its contains cycles. From theses characteristics it will determine if the graph is a tree or not.

A. Program Documentation

The program is only compatible with any recent Windows distribution. In order to run you must first compile the source code, refer to Appendix A. This has to be done using a C++ compiler, it is recommended that you use the Visual Studio 2017 version in order to avoid any errors. Upon execution of the program a console window will pop-up. This console window will serve as the user interface of the program. Entering the Vertices of a Graph

- Vertices must be entered in $\{a,b,c,d\}$ form as stated in the console. Where the characters such as a,b,c,d are the vertices of the graph. It is required that the vertices be one character, if more than one character is inputted the error bellow will be outputted through the command window.

  ERROR: YOU CAN ONLY INPUT SINGLE CHARACTERS FOR VERTICES

- Edges must be entered in $\{ab,ac\}$ where the edge $\{ab\}$ is the link between the first vertex a and the second vertex b. The only requirement is to make sure to enter the correct character for the corresponding vertices inputted.

- If requirements are met the result will be one of the following.

  The graph is disconnected and contains cycles, therefore graph is not a tree.

  The graph is connected and contains cycles, therefore graph is not a tree.

  The graph is disconnected and acyclic, therefore graph is not a tree.

  The graph is connected and acyclic, therefore graph is a tree.

IV. ALGORITHM AND SYSTEM USED

In order to develop an application to analyze a graph, a system would first have to be developed in order to translate any given graph into data. The $V(G)$ and $E(G)$ notation was the preferred notation since it is self-explanatory and nearly universally understood. Once the preferred notation was translated into data we can process the data through the algorithm that determines whether the graph is a tree or not.

A. Translating Graph Notation into Data

The data necessary to process a graph and determine whether it's a tree or not is the number vertices and the adjacent vertices of the given vertices. In order to do this we must format the user input for vertices and edges.

```
void formatVertices(string vertices, vector<string> &verticesData) {
 string vt; //Used as placeholder to assign vertices to
 int lcheck = 0; //letter check used to ensure character per vertex is 1
 for (int i = 0; i < vertices.length(); i++) {
  if (vertices.substr(i, 1) == "}") {
   verticesData.push_back(vt); //Add vertex to vector
   vt.clear(); //Clear string placeholder
  }
  else if (vertices.substr(i, 1) == ",") {
   verticesData.push_back(vt); //Add vertex to vector
   vt.clear(); //Clear string placeholder
   lcheck = 0; //Reset character count to 0
  }
  else {
   vt = vertices.substr(i, 1); //Set vt equal to point in vertex string
   lcheck++; //Increase character count by 1
  }
  if (lcheck > 1) { //Character check
   i = vertices.length() + 1;
   cout << "ERROR: YOU CAN ONLY INPUT SINGLE CHARACTERS FOR VERTICES" << endl;
   terminate();
  }
 }
}
```

Listing 1: Formatting Vertices

Using C++, with the user input in $V(G) = \{a, b, c\}$ notation for vertices we can assign the user input to a string. Using vector data to format the string serves advantageous since vector data is not constrained to an initial length such as an array. By running a loop that runs for the length of the string we can use the substr function to assign every character to the vector. With the vector now containing the vertices as objects we can easily check for the number of vertices by using a function of vectors called length() that returns the number of objects in a vector. This given procedure can be seen in the function specified in **Listing 1**.

```
void fEdges(vector<string> verticesData, vector<int> *adj,string edges) {
 int o, t, c = 0;
 //Go through edge string
 for (int i = 0; i < edges.length(); i++) {
  //Compare edge string to vertices
  for (int j = 0; j < verticesData.size(); j++) {
   if (verticesData[j] == edges.substr(i,1)) {
    //Set 1st match of edge equal to o
    if (c == 0) {
     o = j;
     c++;
    }
    //Set 2nd match of edge equal to t
    else {
     t = j;
     c++;
    }

   }
  }
  //Pushing o and t to each others adjacent vector
  if (c == 2) {
   adj[o].push_back(t);
   adj[t].push_back(o);
   c = 0;
  }
 }
}
```

Listing 2: Formatting Edges into Adjacency Vector

The processing of the edges will also be done using vector data for the convenience of not having a specified length. We will also assign the user input for the edge which is in $E(G) = \{ab, ac\}$ notation to a string. Now with the use of loops we can conveniently compare every character in the edges string to the vertices vector and assign them to their specified adjacent vector. Instead of having strings and the vector being one dimensional the adjacency vector we will have numbers and be two dimensional, where columns 0 1 2 are the used as the adjacent vertices for the vertice in the inputted order. For instance adjacent[0][0] will show the first adjacent vertex found for the first vertex and adjacent[1][1] will show the second adjacent vertex found for the second vertex. Now we simply use an algorithm to correctly assign the adjacent vertices. When we find that a character in the edges string is equal to a vertex we assign it to a place holder integer o that is the vertex number and increase character count by 1. Once another vertex is found we check the character count and if it is not equal to 1 we assign the 2nd vertex to place holder integer t and increase character count by 1. With a check of character count equal to 2 we assign the adjacent vertices to their prospective adjacent vectors and use the push_back() function to add the edge to the adjacent vector while also resetting character count equal to 0. The formatting edge function can be seen in **Listing 2**.

B. Processing Graph Data to Conclude Tree Characteristic

As discussed in section **II** in order for a graph to be a tree it must be acyclic and connected. So in order to have our application determine if the inputted graph is a tree we must have three functions, first to determine if it contains cycles, second to determine if it's connected or disconnected and third to determine if it's a tree. We can easily combine the first two functions by recursively checking adjacent vertices for cycles and marking them as visited, if any vertex is not marked as visited then it must not have an edge to any other vertex therefore it is disconnected.

```
bool cycleCheck(int v,bool visited[], int parent, vector<int> *adj)
{
 visited[v] = true; //mark vertex as visited
 //Go through all vertices adjacent to vertex
 vector<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
  //If not visited recursively check adjacent vertices of vertex for cycles
  if (!visited[*i])
  {
   if (cycleCheck(*i, visited, v,adj)) {
    return true;
   }
  }
  //If adjacent is visited and not parent of the current vertex
  //Then there is a cycle.
  else if (*i != parent) {
   return true;
  }
 }
 return false;
}
```

Listing 3: Function to Determine if Graph contains Cycles

To check for cycles we must recursively run through every adjacent vertex starting from initial vertex 0 in the adjacent vector. As we go down every adjacent vertex and its adjacent vertices and so forth we mark them as visited. If through any recursive search for any adjacent vertices of a vertex we encounter an already visited vertex we categorize the graph as containing cycles by returning true for the cycleCheck boolean. The given algorithm can be seen in **Listing 3**.

```
bool isTree(vector<string> verticesData, vector<int> *adj)
{
 int cycle = 0,con = 0;
 //Initialize vertices to not visited
 bool *visited = new bool[verticesData.size()];
 for (int i = 0; i < verticesData.size(); i++)
  visited[i] = false;
 //Run cycleCheck from vertex 0
 if (cycleCheck(0, visited, -1, adj)) {
  cycle = 1;
 }
 //Do connected and disconnected check by checking for unvisited vertex
 for (int u = 0; u < verticesData.size(); u++)
  if (!visited[u]) {
   con = 1;
  }
 //Output Result
 if (con == 1 && cycle == 1) {
  cout << "The graph is disconnected and contains cycles, ";
  return false;
 }
 if (con == 1 && cycle == 0) {
  cout << "The graph is disconnected and acyclic, ";
  return false;
 }
 if (con == 0 && cycle == 1) {
  cout << "The graph is connected and contains cycles, ";
  return false;
 }
 if (con == 0 && cycle == 0) {
  cout << "The graph is connected and acyclic, ";
  return true;
 }
}
```

Listing 4: Function to Determine if Graph is Tree

Determining if a graph is a tree is much simpler since we have already programmed a function to check for cycles and implemented a way to determine if it's connected or not. Now we simply have to call the function cycleCheck and check if any vertex was not visited. With the results we can output the attributes respectively and determine if the graph is a tree or not.

V. TESTING APPLICATION WITH TEST CASES

In order to establish whether or not the application accurately depicts if a graph is a tree or not we must input various graphs with distinct characteristics. Since there are four possible results we will use two graphs for every result making it eight test. We will compare the programs output to the actual characteristics of the graph and conclude if it is accurate.

A. Not a Tree: Disconnected and Contains Cycles

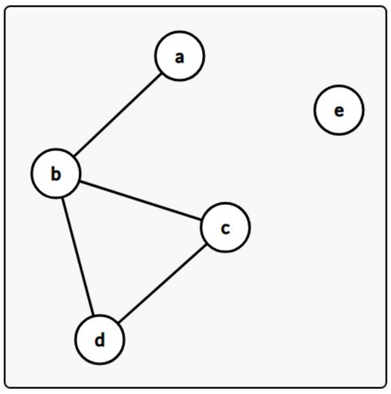
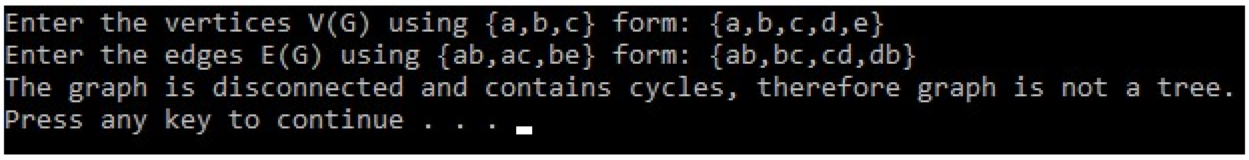


FIG. 4: Example 1

$V(G) = \{a,b,c,d,e\}$
$E(G) = \{ab,bc,cd,db\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d,e}
Enter the edges E(G) using {ab,ac,be} form: {ab,bc,cd,db}
The graph is disconnected and contains cycles, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 5: Output of Example 1

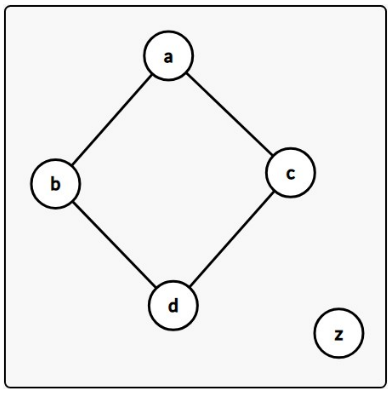
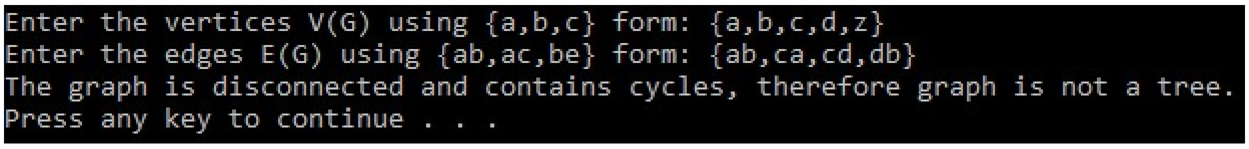


FIG. 6: Example 2

$V(G) = \{a,b,c,d,z\}$
$E(G) = \{ab,ca,cd,db\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d,z}
Enter the edges E(G) using {ab,ac,be} form: {ab,ca,cd,db}
The graph is disconnected and contains cycles, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 7: Output of Example 2

B. Not a Tree: Disconnected and Acyclic

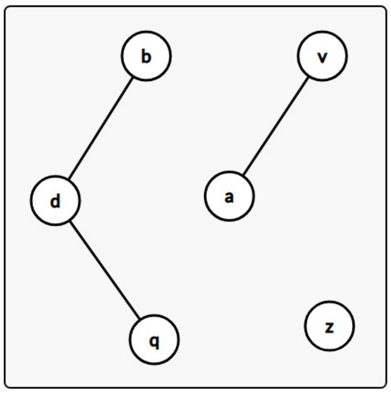
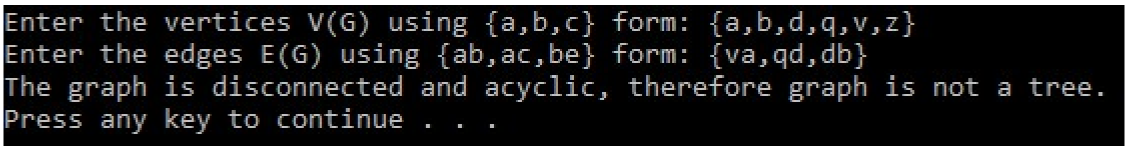


FIG. 8: Example 3

$V(G) = \{a,b,d,q,v,z\}$
$E(G) = \{va,qd,db\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,d,q,v,z}
Enter the edges E(G) using {ab,ac,be} form: {va,qd,db}
The graph is disconnected and acyclic, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 9: Output of Example 3

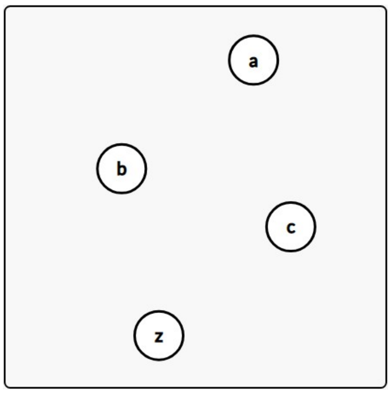
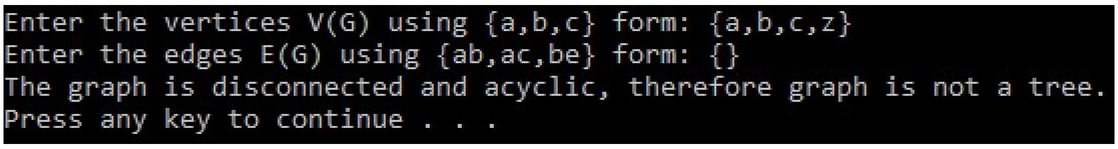


FIG. 10: Example 4

$V(G) = \{a,b,c,z\}$
$E(G) = \{\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,z}
Enter the edges E(G) using {ab,ac,be} form: {}
The graph is disconnected and acyclic, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 11: Output of Example 4

C. Not a Tree: Connected and Contains Cycles

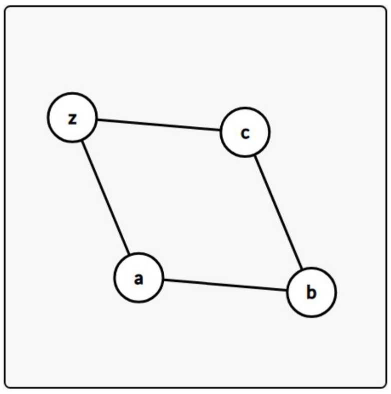
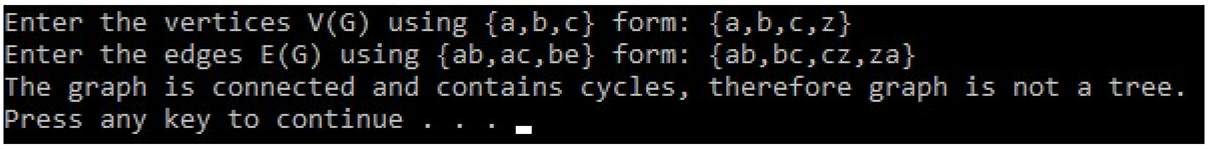


FIG. 12: Example 5

$V(G) = \{a,b,c,z\}$
$E(G) = \{ab,bc,cz,za\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,z}
Enter the edges E(G) using {ab,ac,be} form: {ab,bc,cz,za}
The graph is connected and contains cycles, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 13: Output of Example 5

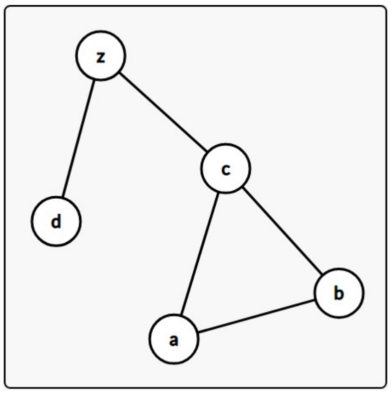
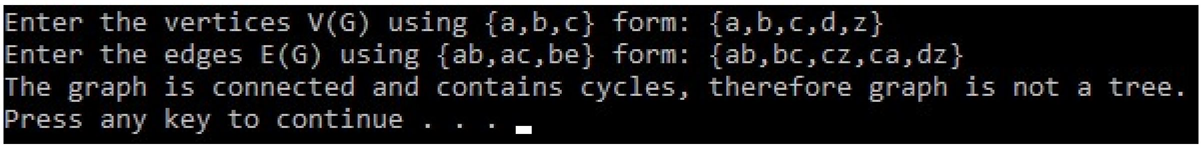


FIG. 14: Example 6

$V(G) = \{a,b,c,d,z\}$
$E(G) = \{ab,bc,cz,ca,dz\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d,z}
Enter the edges E(G) using {ab,ac,be} form: {ab,bc,cz,ca,dz}
The graph is connected and contains cycles, therefore graph is not a tree.
Press any key to continue . . .
```

FIG. 15: Output of Example 6

D. Tree: Connected and Acyclic

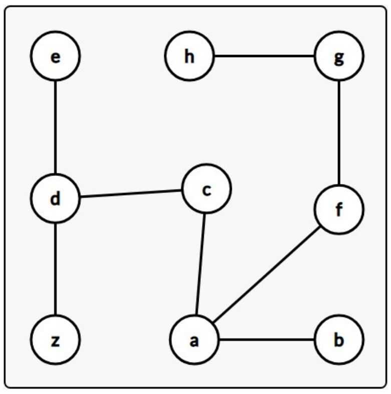
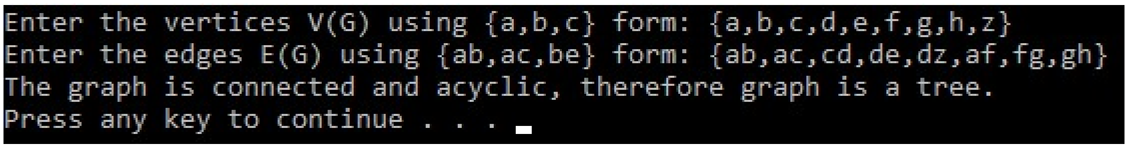


FIG. 16: Example 7

$V(G) = \{a, b, c, z\}$
$E(G) = \{ab, bc, cz, za\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d,e,f,g,h,z}
Enter the edges E(G) using {ab,ac,be} form: {ab,ac,cd,de,dz,af,fg,gh}
The graph is connected and acyclic, therefore graph is a tree.
Press any key to continue . . .
```

FIG. 17: Output of Example 7

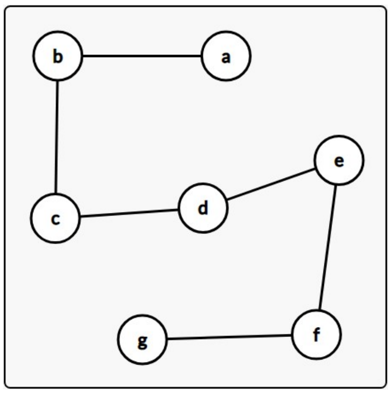
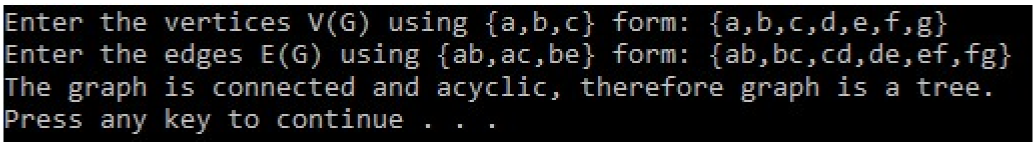


FIG. 18: Example 8

$V(G) = \{a, b, c, d, e, f, g\}$
$E(G) = \{ab, bc, cd, de, ef, fg\}$



```
Enter the vertices V(G) using {a,b,c} form: {a,b,c,d,e,f,g}
Enter the edges E(G) using {ab,ac,be} form: {ab,bc,cd,de,ef,fg}
The graph is connected and acyclic, therefore graph is a tree.
Press any key to continue . . .
```

FIG. 19: Output of Example 8

E. Analysis of Test Results

Upon analysis of all eight test results for the graph, the actual characteristics and predicted characteristics by the program are identical. From the data we can conclude that the application works with no margin of error and will accurately depict whether a graph is a tree or not.

VI. CONCLUSION

In conclusion the application can be used to accurately calculate whether a user inputted graph is a tree or not. The only limitations set by the program are the number of characters available to the user. Future improvements of the program can be removing this limitation and allowing vertices with more than one letter as its respective name. Although this might complicate the notation for the edges, so an overhaul of the processing of the graph data would be needed. Regardless the application can be used to conclude if complex graphs are trees or not. Further improvements can be done to the application by adding more features such as characterizing a graph as Eulerian and Hamiltonian.

APPENDIX A: SOURCE CODE

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

void formatVertices(string, vector<string>&);
void formatEdges(string, vector<string>&);
void formatAdjacents(vector<string>, vector<string>, vector<vector<string>>&);
void fEdges(vector<string>, vector<int>*, string);
bool cycleCheck(int, bool[], int, vector<int>*);
bool isTree(vector<string>, vector<int>*);


void main() {
 string vertices, edges;
 vector<string> verticesData;
 vector<string> edgesData;
 vector<vector<string>> adjacentData;
 vector<int> *adj;
 cout << "Enter the vertices V(G) using {a,b,c} form: {";
 getline(cin, vertices);
 formatVertices(vertices, verticesData);
 adj = new vector<int>[verticesData.size()];

 cout << "Enter the edges E(G) using {ab,ac,be} form: {";
 getline(cin, edges);
 fEdges(verticesData,adj,edges);
 if (isTree(verticesData, adj)) {
  cout << "therefore graph is a tree." << endl;
 }
 else {
  cout << "therefore graph is not a tree." << endl;
 }
 system("pause");
}

void formatVertices(string vertices, vector<string> &verticesData) {
 string vt; //Used as placeholder to assign vertices to
 int lcheck = 0; //letter check used to ensure character per vertex is 1
 for (int i = 0; i < vertices.length(); i++) {
  if (vertices.substr(i, 1) == "}") {
   verticesData.push_back(vt); //Add vertex to vector
   vt.clear(); //Clear string placeholde
  }
  else if (vertices.substr(i, 1) == ",") {
   verticesData.push_back(vt); //Add vertex to vector
   vt.clear(); //Clear string placeholder
   lcheck = 0; //Reset character count to 0
  }
  else {
   vt = vertices.substr(i, 1); //Set vt equal to point in vertex string
   lcheck++; //Increase character count by 1
  }
  if (lcheck > 1) { //Character check
   i = vertices.length() + 1;
   cout << "ERROR: YOU CAN ONLY INPUT SINGLE CHARACTERS FOR VERTICES" << endl;
   terminate();
  }

 }
}

void fEdges(vector<string> verticesData, vector<int> *adj,string edges) {
 int o, t, c = 0;
 //Go through edge string
 for (int i = 0; i < edges.length(); i++) {
  //Compare edge string to vertices
  for (int j = 0; j < verticesData.size(); j++) {
   if (verticesData[j] == edges.substr(i,1)) {
    //Set 1st match of edge equal to o
    if (c == 0) {
     o = j;
     c++;
    }
    //Set 2nd match of edge equal to t
    else {
     t = j;
     c++;
    }

   }
  }
  //Pushing o and t to each others adjacent vector
  if (c == 2) {
   adj[o].push_back(t);
   adj[t].push_back(o);
   c = 0;
  }
 }
}

bool cycleCheck(int v,bool visited[], int parent, vector<int> *adj)
{
 visited[v] = true;
 vector<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
  if (!visited[*i])
  {
   if (cycleCheck(*i, visited, v,adj)) {
    return true;
   }
  }
  else if (*i != parent) {
   return true;
  }
 }
 return false;
}

bool isTree(vector<string> verticesData, vector<int> *adj)
{
 int cycle = 0,con = 0;
 //Initialize vertices to not visited
 bool *visited = new bool[verticesData.size()];
 for (int i = 0; i < verticesData.size(); i++)
  visited[i] = false;
 //Run cycleCheck from vertex 0
 if (cycleCheck(0, visited, -1, adj)) {
  cycle = 1;
 }
 //Do connected and disconnected check by checking for unvisited vertex
 for (int u = 0; u < verticesData.size(); u++)
  if (!visited[u]) {
   con = 1;
  }
 //Output Result
 if (con == 1 && cycle == 1) {
  cout << "The graph is disconnected and contains cycles, ";
  return false;
 }
 if (con == 1 && cycle == 0) {
  cout << "The graph is disconnected and acyclic, ";
  return false;
 }
 if (con == 0 && cycle == 1) {
  cout << "The graph is connected and contains cycles, ";
  return false;
 }
 if (con == 0 && cycle == 0) {
  cout << "The graph is connected and acyclic, ";
  return true;
 }
}
```