Octave Graph Theory Toolbox

Amanda M. Olsen

April 27, 2010

Abstract

This research examines possible ways of determining properties of and manipulating graphs in Octave. Graphs are found in every aspect of life, ranging from a graph of flights among airports in the United States to a network of cell phone towers. It is imperative that we be able to understand the connectivity in the these graphs and other characteristics along with how to alter the graphs in the event of a change. Octave is a compatible programming language for graphs since its primary data structure is a matrix and all graphs can be represented as a matrix. I created eleven codes for Octave that will perform simple characterizations and manipulations of graphs in Octave using the adjacency matrices of the graphs. Sample codes in Octave were analyzed and then expounded upon to arrive at the codes for graph theory applications. The result of this project is the beginnings of a rather extensive graph theory toolbox for Octave that would be able to analyze all types of graphs and apply graph theory algorithms to any graph.

Contents

1 Introduction……...3

2 Basic Characteristics of a Graph……...4

3 Load Graph……...6

4 Get Order and Size……7

5 Labeling Vertices……..8

6 Get Edge……9

7 Set Edges in a Simple Graph……..11

8 Delete a Vertex……...13

9 Delete an Edge……15

10 Clear Graph……...17

11 Specific Graphs……18

11.1 Complete Graph……18

11.2 Create an Empty Graph……..20

12 Conclusion……21

1 Introduction

Graph theory, the study of graphs, is field of mathematics that has applications in network analysis, chemistry, physics, sociology, and biology. With these broad applications of graph theory, it would be useful to perform quick analyses of graphs. With the rise of computers, we can accomplish this task. Some programming languages have already begun to create graph theory related functions. However, there are other languages do not have any of these functions.

Octave is a high-level programming language used primarily for numerical computations. It is an open source language whose primary data structure is a matrix, much like the programming language MATLAB. Since graphs can be represented as matrices, Octave is a natural fit for graph theory functions. Despite this compatibility, there does not exist a toolbox of graph theory codes for Octave. Thus, the main objective of my research is to create a collection of codes for the programming language Octave that will allow us to perform a number of tasks needed in the field of graph theory.

2 Basic Characteristics of a Graph

We first introduce some graph theory terminology.

- A *graph* G is a triple consisting of a vertex set V (G), an edge set E(G), and a relation that associates with each edge two vertices (not necessarily distinct) called its endpoints.
- A *loop* is an edge whose endpoints are equal. *Multiple edges* are edges having the same pair of endpoints.
- A *simple graph* is a graph having no loops or multiple edges.

- The *order* of a graph G, written n(G), is the number of vertices in G. An *n-vertex graph* is a graph of order n. The *size* of a graph G, written e(G), is the number of edges in G. For n 2 N, the notation [n] indicated the set {1,..., n}.
- Let G be a loopless graph with vertex set V (G) = {v_{1},…, v_{n}} and edge set E(G) = {e_{1},…, e_{n}}. The *adjacency matrix* of G, written A(G), is the n-by-n matrix in which entry a_{i,j} is the number of edges in G with endpoints {v_{i}, v_{j}}.

We will consider Graph G in Figure (1) below which is a representation of departing flights from selected airports with destinations to one of the other selected airports during a specific thirty minute period.

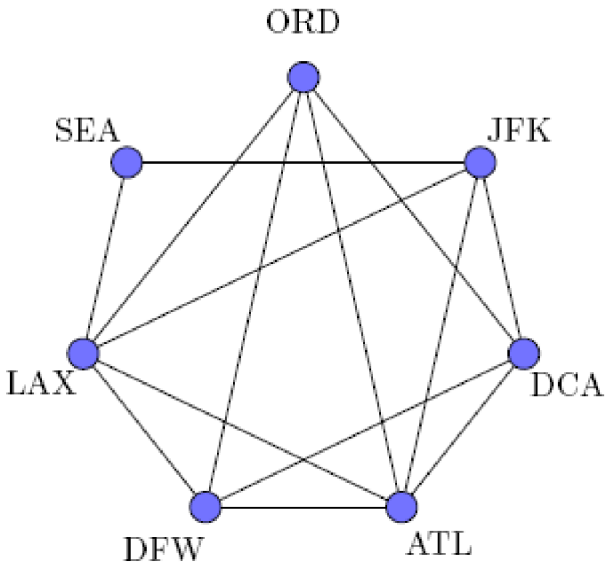

Figure 1: Airport Example

Graph G has the vertex set

V (G) = {ATL, DCA, DFW, JFK, LAX, ORD, SEA}

and the edge set,

E(G) = {(ATL, DCA), (ATL, DFW), (ATL, JFK), (ATL, LAX), (ATL, ORD), (DCA, DFW), (DCA, JFK), (DCA, ORD), (DFW, LAX), (DFW, ORD), (JFK, LAX), (JFK, SEA), (ORD, LAX), (LAX, SEA)}

The order of Graph G is seven since there are seven vertices, and the size of G is fourteen since there are fourteen edges in the graph. G is a simple graph since there are no loops or multiple edges. The adjacency matrix for G is given as

$$\begin{array}{c} \\ ATL \\ DCA \\ JFK \\ ORD \\ SEA \\ LAX \\ DFW \end{array} \begin{array}{c} \begin{array}{ccccccc} ATL & DCA & JFK & ORD & SEA & LAX & DFW \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

3 Load Graph

In a several cases, there is an adjacency matrix already created and saved in a .data-_le compatible to the chosen programming language. Therefore, the adjacency matrix needs to be loaded into the memory space. To accomplish this, we use the load graph function. A detailed description of the workings of the function are given here.

load graph(`str')

- Loads a *.mat-file* whose name is input as a string that contains an adjacency matrix and stores it in a struct (a data structure that characterizes an object by names of qualities).
- Calls the first field of the struct which is the adjacency matrix.
- The output is given by the adjacency matrix.

Let *airport.mat* be the *.mat-file* that contains the adjacency matrix for Graph G in Figure (1). Then

F = load_graph(`airport')

would set F equal to the adjacency matrix given in the previous section.

4 Get Order and Size

After loading the adjacency matrix with the load graph function, it is necessary to find the order and size of the graph it represents. We do this by using the function that determines the order and size by the adjacency matrix. The workings of this function are given below.

get_order_size(A)

- Takes the adjacency matrix A as the input.
- Sets the variable rows to the number of rows in A and the variable cols to the number of columns in A.
- Sets variable n to the value for rows.
- *for-loop* sums all the upper triangular entries in A and sets this value to the variable e.
- The output is given by a struct with fields `adj' set to A, `order' set to n, and `size' set to e.

Since we set the adjacency matrix for graph G in Figure (1) to F,

G = get_order_size(F)

would create a struct G where the field `adj' is the adjacency matrix for the graph of the airport example in Figure (1), the field `order' is set to seven since there are seven vertices in the graph, and the field `size' is set to fourteen since there are fourteen edges in the graph.

5 Labeling Vertices

After creating the struct that contains the adjacency matrix, order, and size of the graph, we need to label the vertices. In the airport example whose graph is given in Figure (1), we need to label vertex 1 as ATL, vertex 2 as DCA, and so forth, since ATL corresponds to the first row and column of the adjacency matrix, DCA corresponds to the second row and column, and so forth. To accomplish this we utilize the labeling vertices function. The workings of the function are given by

vertex_label(G)

- Takes the struct G with fields `adj', `order', and `size' as the input.
- Sets variable n to the value in G.order.
- Creates a cell of size nx1.
- *for-loop* outputs to screen `Input the label for vertex n' for all values from 1 to n.
- User then inputs labels for each vertex as either strings or numerical values.
- The output is given by the struct G with the additional field `label' set to the cell created in the for-loop.

Applying this function to Graph G in Figure (1), we input

G = vertex_label(G)

since, in the previous section, we declared G to be the struct representing our graph. Notice that since we set G to be vertex label(G), we are renaming G to be the struct that is output after the completion of the function. Thus, the struct G input into the function will no longer exist. Now the output to screen would then be `Input the label for vertex 1', and we would input `ATL'. Then the next output would be `Input the label for vertex 2', and we would input `DCA'. This would continue to the last vertex where the output would be `Input the label for vertex 7', and we would input `DFW'. Once this is complete, the struct G would be modified to include a new field `label' where the entries are contained in a 7x1 cell given below.

$$\begin{pmatrix} \text{'ATL'} \\ \text{'DCA'} \\ \text{'JFK'} \\ \text{'ORD'} \\ \text{'SEA'} \\ \text{'LAX'} \\ \text{'DFW'} \end{pmatrix}$$

6 Get Edge

Often it is important to know whether two vertices are adjacent. In selecting flights, flyers often want to make the flight have the least amount of connections possible with a direct flight being optimal. Thus, flyers would be concerned with locating whether a flight exists between two airports. In the graph of the

airport example in Figure (1), say there is a flyer looking for a flight between ATL and ORD and another flyer looking for a flight between DCA and LAX. We now need a way to determine whether this is true. This is done by using the get edge function whose workings are given below.

get_edge(G,`vertexlabel1',`vertexlabel2')

- Takes the struct G with necessary fields `adj' and `label'.
- Sets u and v to the row values for `vertexlabel1' and `vertexlabel2' in the field `label', respectively.
- The output is given by the (u, v)th entry of the adjacency matrix in the field `adj'.

Now we solve the dilemma for the two flyers that need flights between ATL and ORD and between DCA and LAX. For the first flyer, we input

a = get_edge(G,`ATL',`ORD').

The function will locate the entry between the vertices `ATL' and `ORD', or vertices 1 and 4, above the diagonal in the adjacency matrix in the struct G, which is the adjacency matrix of the graph of the airport example in Figure (1). In the adjacency matrix provided again below, we can clearly see that the output is 1, which implies that there is a flight between `ATL' and `ORD'. Thus, this flyer will be a satisfied customer to have located a direct flight.

$$
\begin{array}{c} \\ ATL \\ DCA \\ JFK \\ ORD \\ SEA \\ LAX \\ DFW \end{array}
\begin{array}{c}
\begin{array}{ccccccc} ATL & DCA & JFK & ORD & SEA & LAX & DFW \end{array} \\
\left(\begin{array}{ccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0
\end{array} \right)
\end{array}
$$

Now we need to help the second flyer. This time we input

a = get_edge(G,`DCA',`LAX').

As before, the function will locate the entry between the vertices `DCA' and `LAX', or vertices two and six, above the diagonal in the adjacency matrix of the graph of the airport example in Figure (1). This time the output is 0, implying that there is no flight between `DCA' and `LAX'. Thus, unfortunately this flyer will be forced to have at least one connecting flight to get from `DCA' to `LAX'.

7 Set Edges in a Simple Graph

For the unfortunate situation of the second flyer, the airline might consider trying to schedule a flight that would connect `DCA' to `LAX' so that this flyer will be more inclined to fly again. So the graph would need to be altered to include a new edge between `DCA' and `LAX'. This also means that the adjacency matrix and the size of the graph would need to be adjusted. We achieve this by using the set edge unweighted function whose workings are given below.

set_edge_unweighted(G,`vertexlabel1',`vertexlabel2')

- Takes the struct G with the necessary fields `adj' and `label'.
- Utilizes set_edge(G,`vertexlabel1',`vertexlabel2',1) to do the following:
    - Sets u and v to the row values for `vertexlabel1' and `vertexlabel2' in the field `label' respectively.
    - Sets the (u, v)th entry to one and the (v, u)th entry to 1 in the adjacency matrix in the field `adj'.
    - Calculates order and size using get order size(G.adj).
- Outputs the struct G with the adjusted field `adj' and the fields `order', `size', and `label'.

There is a distinction between set edge and set edge unweighted since there are weighted and unweighted graphs. For the purpose of this paper, we will not discuss weighted graphs. We are dealing strictly with unweighted graphs which means that every entry in the adjacency matrix of any graph will either be a 0 to indicate two vertices are not adjacent or a 1 to indicate two vertices are adjacent.

Consider the previously proposed objective of adding an edge between `DCA' and `LAX'. We would input

H = set edge_unweighted(G,`DCA',`LAX')

to create a new struct H that will add an edge between `DCA' and `LAX' in the adjacency matrix in the struct G representing the graph of the airport example in Figure (1). The function will locate the entries both above and below the diagonal in the adjacency matrix of the graph of the airport example in Figure (1) that corresponds to the intersection of the vertices `DCA' and `LAX', or vertices two and six. Now the entries will be set to 1 and the new adjacency matrix would be given by

$$
\begin{array}{c}
 \\
ATL \\
DCA \\
JFK \\
ORD \\
SEA \\
LAX \\
DFW
\end{array}
\begin{array}{c}
\begin{array}{ccccccc}
ATL & DCA & JFK & ORD & SEA & LAX & DFW
\end{array} \\
\left(\begin{array}{ccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}\right)
\end{array}
$$

The order of the graph would remain the same, but the size of the graph would be increased by one to a total of fifteen since we added an edge to the original graph. So the graph would now appear as

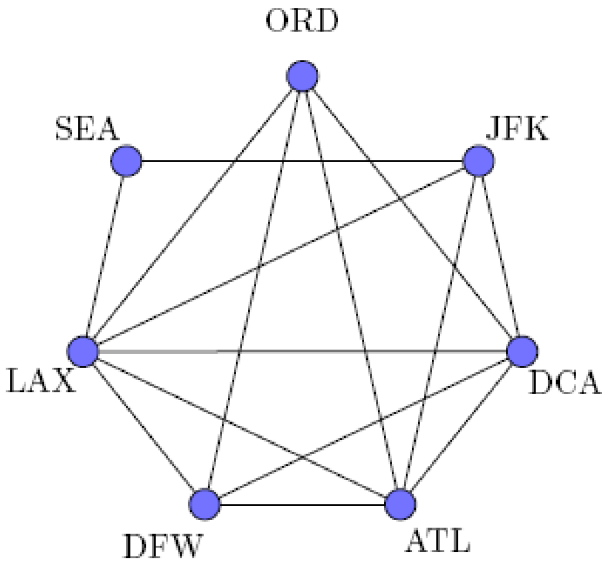

Figure 2: Airport Example with Added Edge Between DCA and LAX

8 Delete a Vertex

There may be some circumstances where a vertex needs to be completely removed from the graph. In the case of the graph provided in this paper, there may be an unfortunate incident where an airport must shut down for security reasons. Therefore, the vertex that represents that particular airport must be removed from the graph. To remove this vertex, we utilize the delete vertex function that will adjust the fields in a struct G representing the graph so that the vertex will be removed. A detailed description of the workings of the function is given here.

delete_vertex(G,`vertexlabel')

- Takes the struct G with the necessary fields `adj' and `label', and the vertex label `vertexlabel'.
- Sets v to the row value for the vertex `vertexlabel' in the adjacency matrix of G.
- Sets the cell value for `vertexlabel' to the empty entry.
- Removes the empty entry from the cell.
- Removes row and column v in the adjacency matrix of G.
- Recalculates order and size of adjacency matrix of G using get order size.m.
- The output is given by the struct with fields `adj', `order', `size', `label' with the new adjacency matrix, order, size, and cell respectively.

We know that G is the struct representing the graph of the airport example in Figure (1). Say that the Chicago O'Hare Airport (ORD) is immediately shut down because of an impending threat. Then we would use

H = delete_vertex(G,`ORD')

to create a new struct H that will contain the adjacency matrix of the struct G with the vertex `ORD' removed. Then the new adjacency matrix in the field `adj' of struct H would appear as follows:

$$\begin{array}{c} \\ ATL \\ DCA \\ JFK \\ SEA \\ LAX \\ DFW \end{array} \begin{array}{c} \begin{array}{cccccc} ATL & DCA & JFK & SEA & LAX & DFW \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

The order of the graph would now be six since one of the vertices was removed. Since all the edges that were incident with the deleted vertex have also been removed from the graph, the size of the graph has been decreased from fourteen to ten. Also the cell that contains labels of the vertices does not include ORD in G and the graph would no longer contain ORD. They would appear as

$$\begin{pmatrix} \text{'ATL'} \\ \text{'DCA'} \\ \text{'JFK'} \\ \text{'SEA'} \\ \text{'LAX'} \\ \text{'DFW'} \end{pmatrix}$$

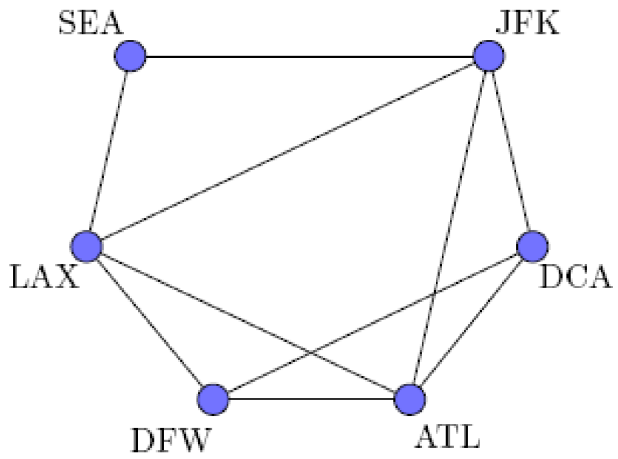

Figure 3: Airport Example with Vertex ORD Removed