2016

GAL- Graph Application Language

COMS4115 SUMMER, PLT

ANTON (AIN2108)
DONOVAN (DC3095)
MACRINA (MML2204)

Introduction:

GAL – Graph Application Language is a language that is intended to facilitate manipulations of graphs. Standard library of GAL will provide the user with a number of functions intended to make it trivial to describe more complicated algorithms.

Motivation:

1. Many real-world problems can be simplified to a graph problem.
2. Many problems can become complex if they are represented using a graph but it is still educational and fun to do so. (Eg. Logic map of mathematics and path from statement A to B as a proof)
3. Implementing the basic graph algorithms will serve as a good revision of these algorithms.
4. Everything is a list in LISP, everything is really an edge in GAL. It is our belief that there is growth to be expected from building a world only from edges.

Language Description:

The language that is proposed in this project is a graph oriented language where the user is able to create a graph structure and manipulate the nodes and edges within the graph to suit any form of topology that the user desires. The language will also enable users to implement any graph algorithm to the created graph, this will not be built into the language but rather a part of the language. In our language, Nodes and Edges will be used as the basic data types and they will have sufficient flexibility that allows the user to create directional, bidirectional or directionless graphs. The language will compile down to LLVM.

Language Functionalities:

- Nodes and Edges are basic data types of the language
- Edges consist of the source, destination nodes and the weight of each edge
- Graph can be defined as a set of nodes, or a set of edges as per the user's preference
- Standard Library functions that will implement basic search and trace functions
- Users will have the ability to create a custom data structure, for example describing a custom queue
- Graphs will be represented as an adjacency list

Sample Code:

The sample code shown below is an implementation of the graphical topologies shown in figures 1 to 3.

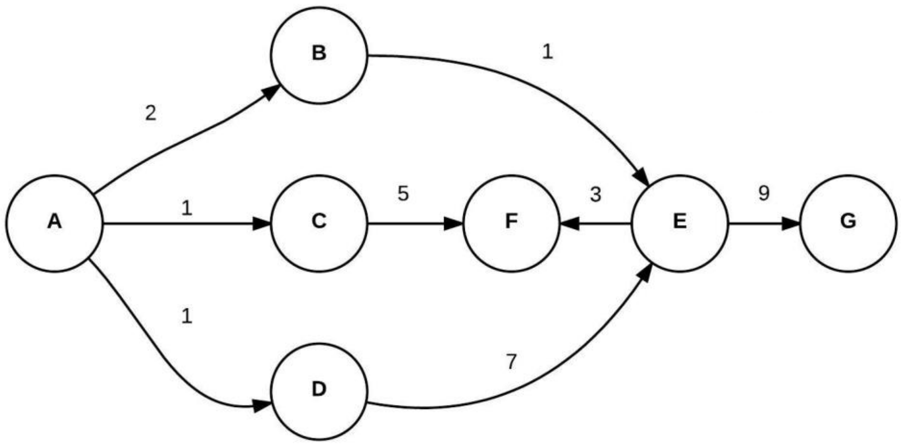

Figure 3: Graph 1

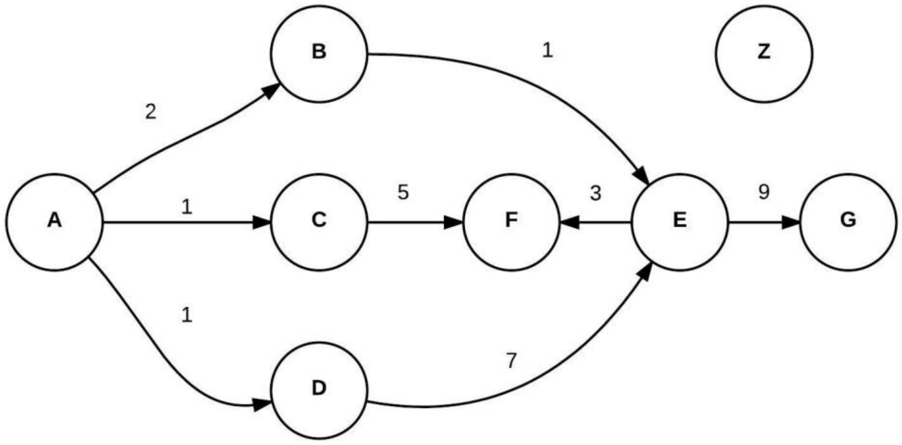

Figure 2: Graph 2

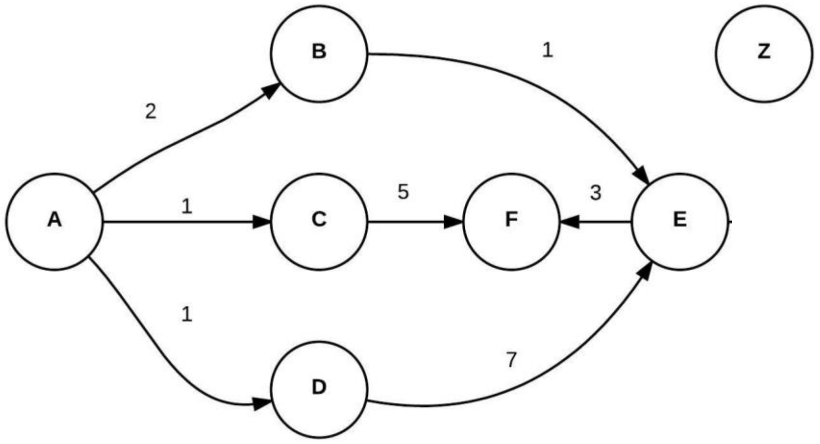

Figure 1: Graph 3

```
//To define the graph in GRAPH 1, we can write:


node n1 = {"A":(2,"B"),(1,"C"),(1,"D")};
node n2 = {"B":(1,"E")};
node n3 = {"C":(5,"F")};
node n4 = {"D":(7,"E")};
node n5 = {"E":(3,"F"),(9,"G")};
graph_node G = (n1, n2, n3, n4, n5);

// Z is a new node

edge e1 = ("A",2,"B");
edge e2 = ("A",1,"C");
edge e3 = ("A",1,"D");
edge e4 = ("B",1,"E");
edge e5 = ("C",5,"F");
edge e6 = ("D",7,"E");
edge e7 = ("E",3,"F");
edge e8 = ("E",9,"G");
graph_edge G = (e1,e2,e3,e4,e5,e6,e7,e8);
// We would also be able to take input from a text file
// To add a new unconnected node "Z" to the graph in GRAPH 1:

G + {"Z"}
// Z is a new node

//To add node "G" to the graph in GRAPH 3:
//add the node via the "node" syntax
G + {"E":(9,"G")}
// "G" gets automatically appended to the node E

// Z is a new node

node n = {"E":(9,"G")};
G + n;


//add the node via the "edge" syntax
G +. ("E",9,"G");

// OR

edge e = ("E",9,"G");
G +.e


//Removing a node from a graph
G - n //removes everything associated with node n from the graph G
G -. e //removes edge 'e' from graph G
int int_data //creates int_data of integer data type
```

Notes about the syntax:

A comment is represented by // or /* */ similar to comments in C

The operator