The *Graph* Programming Language (GPL)

System Architect: Ephraim Donghyun Park (edp2114)
Verification & Validation: Peiqian Li (pl2521)
Manager & Language Guru: Qingxiang Jia (qj2125)

Motivation

Graph is a very powerful data structure that can be used to model a variety of systems in many fields. Graph is such a fundamental model that people have developed libraries dedicated to graphs in almost all general-purpose high-level programming languages. However, implementing graph-related algorithms in languages like Java or C++, even with the benefit of using third-party graph libraries, entails manual manipulation of nodes and edges. This could prove to be error-prone (with pointer manipulations in C++), tedious (verbose especially in Java), and daunting (to people new to the programming world).

Here we propose a domain-specific language that attempts to remedy these problems. The Graph Programming Language (GPL) handles most logic behind implementing graphs under the hood, so that programmers are able to focus more on *using* graphs, instead of *implementing* them.

Proposed Uses

The primary goal of GPL is to allow programmers to create, use, and manipulate graphs in a natural, flexible and intuitive way. All graph-based algorithms should be easier to implement in GPL, e.g. shortest path, spanning tree, strong connectivity. Because all trees *are* graphs, GPL is automatically suitable for applications involving tree structures, such as priority queues (min/max heaps), binary search trees, or any kind of hierarchical data representation.

All variables are type-bound at run time (similar to Python), so there is no need to declare the type of a variable when coding in GPL. A graph can be defined by a list of its edges or a list of neighbours of every node; the syntax for defining graph is designed to be as intuitive as possible. A graph initialized with nodes and edges can be later altered dynamically (e.g. adding an edge, removing an edge and/or nodes, etc). Nodes are not only bound to variables at the time of definition, but also internally indexed in the order they appear in the graph definition. The built-in library supplies common graph algorithms, so developers can take advantage of these instead of having to implement them on their own.

Syntax

1. Data types

1.1 Basic data types

- integer (int)
- floating-point number (float)
- character (char)

1.2 Graph-related types

- node : all node has numerical id, which is unique only within the graph
- edge : all edges are directed edges, bidirectional edge is just a combination of two directed edges. Defined by the two nodes and value.
- graph : all graph

1.3 Other types

- string (str): internally represented by an array of characters
- array: internally represented by a graph, specifically a line graph where each node stores the value of one element in the array

2. Arithmetic, Relational, and Logical Operators

>, <, <=, >=, ==, !=	Basic data type to basic data type -> done by value Node to node -> done by the value of the node Edge to edge -> done by the value of the edge
&&, ||	Logical AND and OR
===, !===	Compares if the two variables refer to the same one.
*	Represent the value of the node when placed in front of node variable.
*, /, +, -	Basic data type to basic data type -> done by value Node to node -> done by the value of the nodes Edge to edge -> done by the value of the edge

3. Control Flow

3.1 Statements and Blocks

;	End a statement.

//	Start of a one-line comment
/*	Start of a comment block
/*	End of a comment block

The comment blocks can also be nested.

3.2 If-Else and Loops

`if (expression) {`
`  ...`
`}`
	if statement. If the expression is simply (i.e. consists only one phrase), the parenthesis and be omitted. If there is only one line, the { } can omitted.

`if (expression) {`
`  ...`
`} else {`
`  ...`
`}`
	if statement can also have an optional else statement. One can also nest the if-else statement.

`for (loop invariants) {`
`  ...`
`}`
	for loop. Omitting rules are the same.

`while (loop invariants) {`
`  ...`
`}`
	while loop. Omitting rules are the same.

4. Miscellaneous Standard Library Functions

`in_degree(node x);`	Returns the number of incoming edges to the node.
`out_degree(node x);`	Returns the number of outgoing edges from the node.
`min_edge(node x);`	Returns the outgoing edge with the minimum value from the node
`max_edge(node x);`	Returns the outgoing edge with the maximum value from the node

`is_strongly_connected(graph x);`	Returns if the graph is strongly connected or not

Program Structure

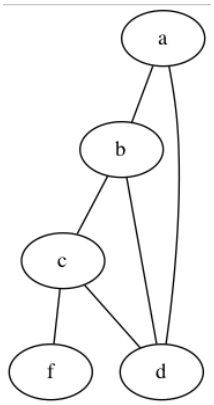
1. **Function Structure**

```
func in_degree(x) {
        // returns the number of incoming edges to the node
}
```

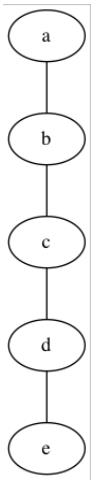
2. **Graph Definition Structure**

Graphs are defined by either listing all edges or adjacency lists of all nodes. Directed edges are indicated by the "->" symbol, whereas undirected edges by "--" (internally stored as two separate edges in opposite directions).

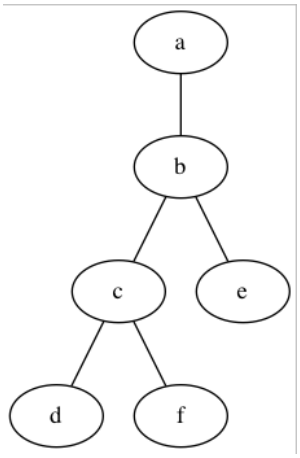
```
g1 = {
    a -- b -- c -- d -- a;
    b -- d;
    c -- f;
}; //list of edges

OR
g1 = {
    a --: b d
    b --: a c d
    c --: b d f
    f --: c
}; //adjacency list
```



```
g2 = {
    a -- b -- c -- d;
};
```




```
g3 = {
    a -- b -- c -- d;
    c -- f;
    b -- e;
};
```



Example Code

Here is an example program that finds the minimum spanning tree of an undirected graph.

```
func min_spanning_tree(g) {
    buf = [] //empty array
    for (e in g.edges()) {
        buf.append( (e.value, e) ); //append the tuple (e.value, e) to buf
    }
    sort(buf); //built-in library function
    total_weight = 0;
    mst = {}; //initialize minimum spanning tree as an empty graph
    for (t in buf) {
```

```
        e = t[1]; //the second element of the tuple is the edge
        if ( e.first in mst.nodes() && e.second in mst.nodes()) {
            //adding this edge would result in a cycle in mst, so we skip
            continue;
        }
        //otherwise, we add this edge to mst
        mst.add(e);
        total_weight += t[0]; //accumulate weight
    }
    return total_weight;
}

func main() {
    g = {
        a -2- b -3- c -4- d -2- a;
        b -5- d;
        c -1- f;
    }; //initialize graph; the number between -- is the weight of that
particular edge
    print(min_spanning_tree(g)); //print the total weight of the MST
}
```