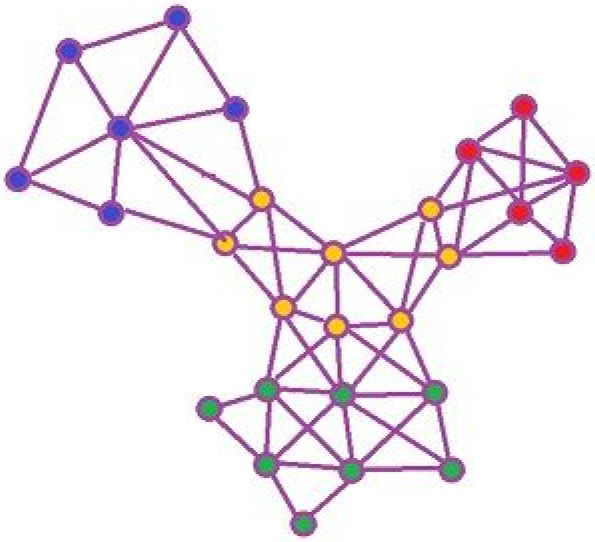
COMS W4115 Programming Languages & Translators

GIRAPHE



Name	UNI
Dianya Jiang	dj2459
Vince Pallone	vgp2105
Minh Truong	mt3077
Tongyun Wu	tw2568
Yoki Yuan	yy2738

Motivation/ Introduction:

Graphs appear naturally in problems of various areas like chemistry, sociology, and many other disciplines. It intuitively expresses entities and complex relationships among them. It is especially prevalent in the fields of computer science, math, and data science where graphs are often used to represent data and social networks.

But there does not seem to exist any computer language that provides the essential features in defining and utilizing graphs. In most case, you need to define a set of vertexes and edges in order to define a graph, which is complicated and not very efficient. We would like to build a language that has many graph-based features which will allow users to effectively define, manipulate, and process their graph data.

Language Description:

GIRAPHE is a graph creation and manipulation language that allows users to define and process graphs specific to their use cases. Using GIRAPHE, most of simple graph (directed or undirected graph, edge with or without values) could be created by a single statement. Several graph operations are also supported in GIRAPHE such as merging two graphs. As we know, most graphs are composed of nodes and lines, and GIRAPHE enables users to define and manipulate graphs in a more convenient and clear way. In addition, GIRAPHE supports plotting graph. Users can plot graphs using “plot” function, in which the process of realization and visualization is simplified.

Design and Syntax:

Primitive Data Types:

Integer	int
Floating point number	float
String	string
Character	char
Boolean	bool
Lists	List, literal: [ ]

Tuple	Tuple<type1,...,typeN>, literal: (5, … ,’A’)
Dict	Dict<valueType>, literal: { }
NULL	null
Void	void

Graph:
Graph - contains Nodes and graph data.
Has methods for basic use, i.e. shortest path, articulation points, merge graphs

Node (interface):
Contains value and list of neighbor nodes
Node - the language will provide a basic class but will also allow the user to implement the interface in order to have Node types specific to their need

Operators:

Basic:

>, <, <=, >=, ==, !=	comparison operators for basic types
!, &&, ||	logical NOT, AND, OR for bool type
+,-,*, /, %	arithmetic operators for int and float
[]	list access
=	assignment operator
;	end of line

Node:

+ Node n1 + Node n2 = Graph g’	Return graph of two nodes
- Node n1 - Node n = Edge e	Return edge between nodes

Statement and Blocks:

;	end of line
//	Start of a one line comment
/*	Start of a comment block
*/	End of a comment block

Control Flow:

if (expression) {
...
}
	if statement. If expression is evaluated to true, the statements between {} will be executed.

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

while (loop condition) {
...
}
	\

do { … } while (loop condition)	\

for (initialization; loop condition; increment;) {
…
}
	\

for (*member* in *collection*) {
…
}
	For-each loop.

continue	\
break	\
return	\

in	\

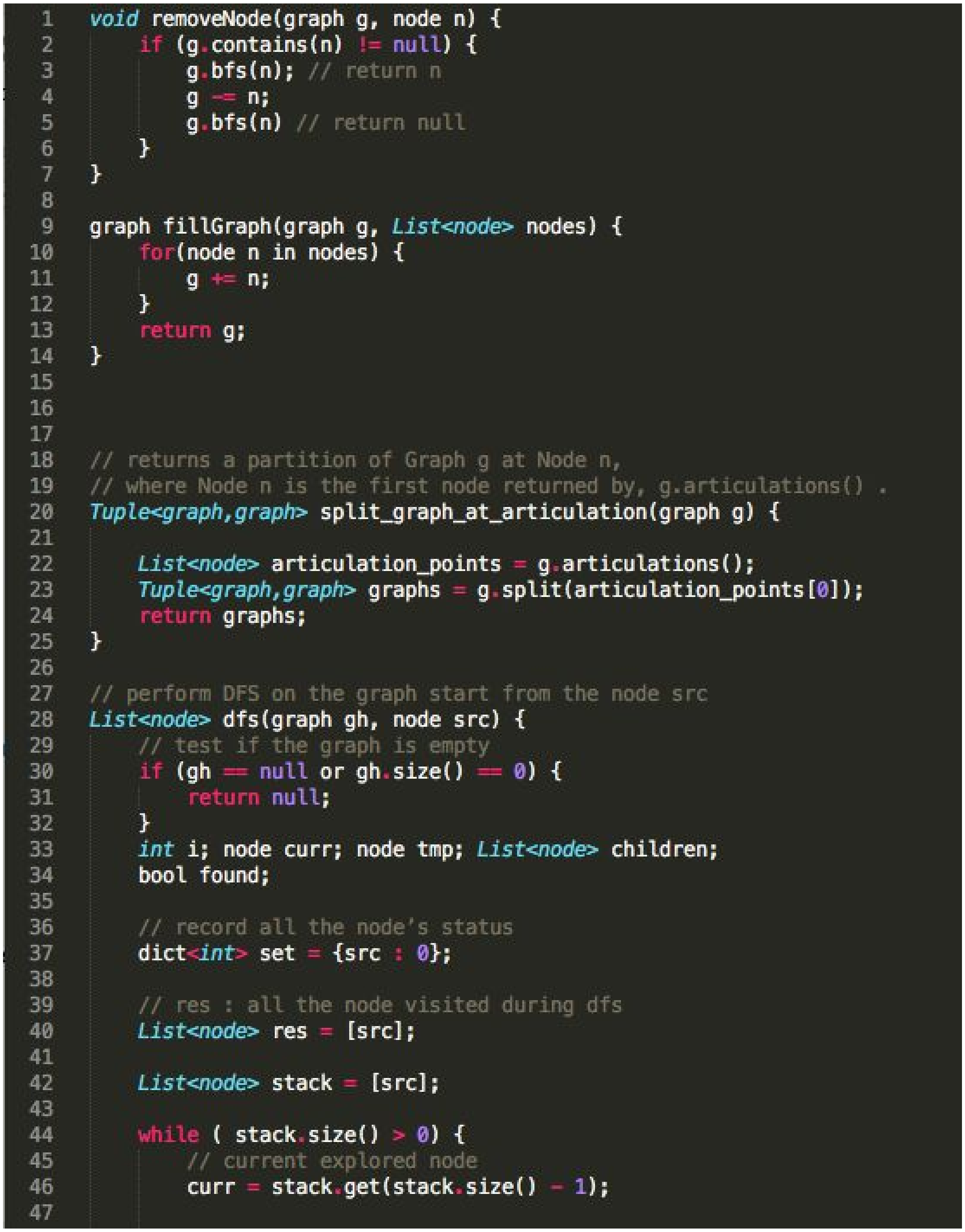
Built-in functions:

API of Graph(G)		
Name	Function Expression	Description
add	G.add(n)	Add node to graph
merge	G.merge(g)	Merge Graph g with Graph G and return a new graph.
contains	G.contains(n)	Check whether node n is in the graph
size	G.size()	Return the number of nodes in G
path	G.path(n1,n2)	Return shortest path between nodes
is empty	G.isEmpty()	Return whether the graph has node or not
remove	G.remove(n)	Remove node from Graph G
get articulation points	G.articulations()	Get articulation points
get a node	G.getNode(int id)	Returns the node with specified id
get all nodes	G.getAllNodes()	returns the list of nodes in the graph
get all edges	G.getAllEdges()	returns the list of edges in the graph
get edge count	G.getEdgeCount()	returns the number of edges in the graph
get node count	G.getNodeCount()	returns the number of nodes in the graph
plot	G.plot()	Plot out the Graph G

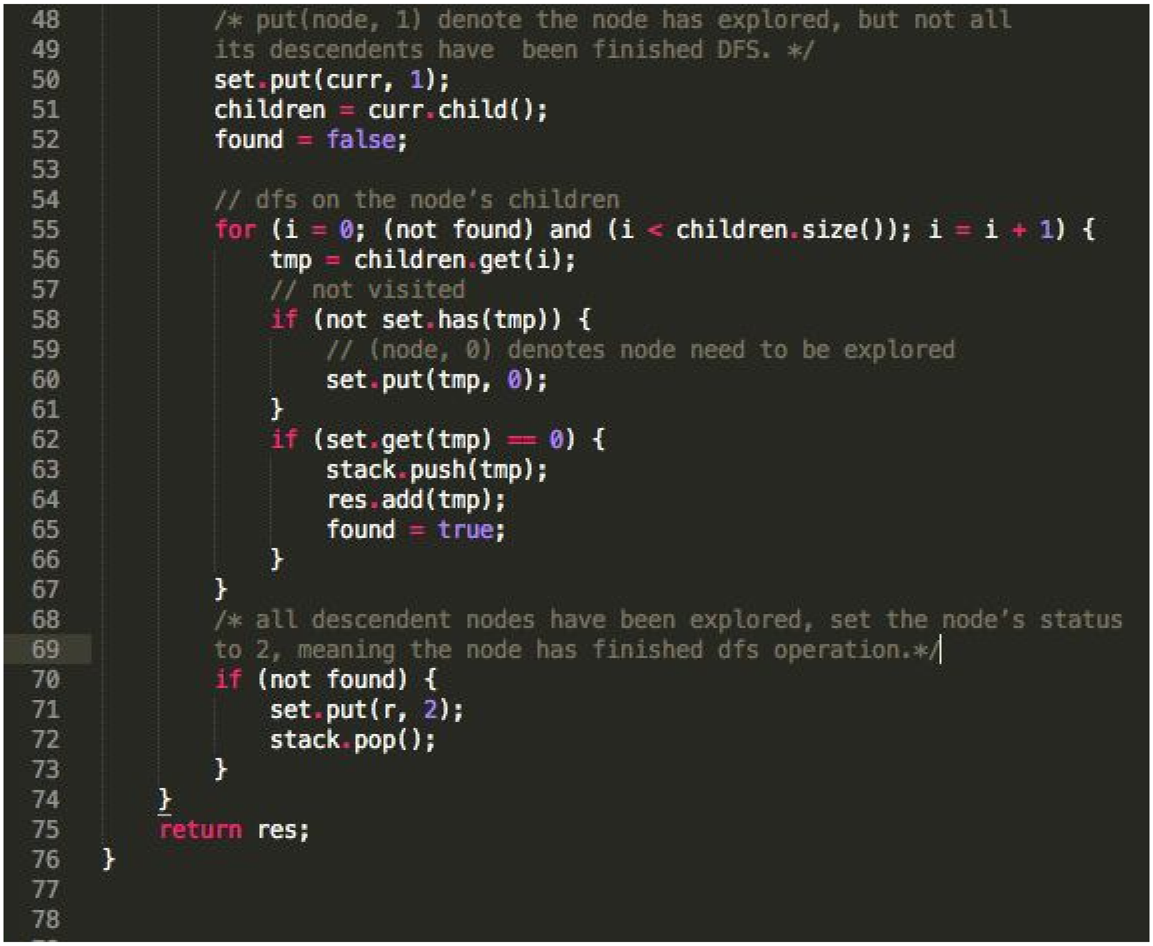
API of Node(N)		
Name	Function Expression	Description
value	N.value()	Return the value of node
neighbor	N.nbr()	Return a list of all connected nodes in an undirected graph
child	N.child()	Return a list of nodes which are children of the current node in a directed graph
parent	N.parent()	Return the parents of a node in a directed graph
getID	N.getID()	Returns the unique integer ID of the specified node

API of Edge(E)		
Name	Function Expression	Description
start	E.start()	Return the starting point of one edge
end	E.end()	Return the end of one edge
weight	E.weight()	Set the weight of the edge
nodes	E.nodes()	Return a list of two nodes connected by the edge E
label	E.label()	Return the boolean value, true if the edge is been labeled
remove	E.remove()	Remove edge E

Example Program:



```
void removeNode(graph g, node n) {
    if (g.contains(n) != null) {
        g.bfs(n); // return n
        g -= n;
        g.bfs(n) // return null
    }
}

graph fillGraph(graph g, List<node> nodes) {
    for(node n in nodes) {
        g += n;
    }
    return g;
}



// returns a partition of Graph g at Node n,
// where Node n is the first node returned by, g.articulations() .
Tuple<graph,graph> split_graph_at_articulation(graph g) {

    List<node> articulation_points = g.articulations();
    Tuple<graph,graph> graphs = g.split(articulation_points[0]);
    return graphs;
}

// perform DFS on the graph start from the node src
List<node> dfs(graph gh, node src) {
    // test if the graph is empty
    if (gh == null or gh.size() == 0) {
        return null;
    }
    int i; node curr; node tmp; List<node> children;
    bool found;

    // record all the node's status
    dict<int> set = {src : 0};

    // res : all the node visited during dfs
    List<node> res = [src];

    List<node> stack = [src];

    while ( stack.size() > 0) {
        // current explored node
        curr = stack.get(stack.size() - 1);

```



```
        /* put(node, 1) denote the node has explored, but not all
        its descendents have  been finished DFS. */
        set.put(curr, 1);
        children = curr.child();
        found = false;

        // dfs on the node's children
        for (i = 0; (not found) and (i < children.size()); i = i + 1) {
            tmp = children.get(i);
            // not visited
            if (not set.has(tmp)) {
                // (node, 0) denotes node need to be explored
                set.put(tmp, 0);
            }
            if (set.get(tmp) == 0) {
                stack.push(tmp);
                res.add(tmp);
                found = true;
            }
        }
        /* all descendent nodes have been explored, set the node's status
        to 2, meaning the node has finished dfs operation.*/
        if (not found) {
            set.put(r, 2);
            stack.pop();
        }
    }
    return res;
}
```