Graphiti

A Simple Graph Language

Language Reference Manual

Alice Thum, System Architect, at3160@barnard.edu
Sydney Lee, System Architect, stl2121@barnard.edu
Michal Porubcin, Language Guru, mp3242@columbia.edu
Emily Hao, Tester, esh2160@columbia.edu
Andrew Quijano, Manager, afq2101@columbia.edu

1. Introduction

1.1 Overview

By convention, Graphiti programs will be saved with a .gra suffix. For example, our upcoming hello world program would be called helloworld.gra. Any code we write will be using the Courier New font, and output will be in blue. Example:

```
print("Hi Everybody");
Hi Everybody
```

1.2 Application

Graphiti is a programming language designed to support graph algorithms. Our language makes it easy for programmers to initialize graphs, add/delete edges and nodes, traverse the graph, and conduct queries. One use case of our language building traditional graphs, from which a user can call upon standard library functions to execute breadth-first search (BFS) and depth-first search (DFS) on a graph.

A second use case is analyzing relationships between two nodes in a graph. One instance in which this is seen is in the Neo4j NoSQL database, which organizes data into nodes and edges which contain labels of relationships. Like Neo4j, our language will be able to build a family tree into a graph and execute a query such as "who is the mother of Jon Snow?", then return the answer based on checking the correct edge. Our language will support edges with either integer weights for classical graph algorithms or relationship names like Neo4j.

2. Data Types

2.1 Data Types Overview

Type	Description
`int`	4 bytes (32 bit) integer values
`double`	8 bytes (64 bit) values
`bool`	1 byte (8 bit) true/false
`char`	1 byte (8 bit) value
`string`	Finite sequence of characters
`list`	Linked list with unordered data. Can support any kind of data
`map`	Collection of key-value pairings of information. Supports <string, string> only
`node`	Struct containing id for the node and additional information
`graph`	A set of nodes and a set of edges that connect a pair of nodes
`null`	Empty value

2.2 Data Types – Simple Usage

Variables are assigned using the = operator. The type of the variable precedes the variable name in declarations, and the value to be assigned is on the right of the = operator.

```
bool z = true;
```

```
int a = 3;
int b = 4;
double d = 3.;
double e = 4.3;
string s = "hello ";
string t = "world!";

int c = a + b;
double f = d + e;
string u = s + t;
```

2.3 Lists

Declaring an empty list:

```
      list l = [];
```

Getting length of a list:

```
      list l = [1, 2, 3, 4];
      int x = l.len();
      x = 4
```

Appending a list:

```
      int z = 5;
      list x = [1, 2, 3, 4];
      l += 5;
      l = l + 6;
      x = [1, 2, 3, 4, 5, 6]
```

Removing head:

```
      list x = [1, 2, 3, 4];
      int z = x.remove_head();
      x = [2, 3, 4]
      z = 1
```

Removing tail:

```
      list x = [1, 2, 3, 4];
      int z = x.remove_tail();
      x = [1, 2, 3]
      z = 4
```

Getting certain element from the list:

```
      list x = [1, 2, 3, 4];
      int y = 2;
      int z = x[y];
      z = 3
```

Adding new head:

```
list x = [1, 2, 3, 4];
int y = 2;
int z = x.add_head(y);
x = [3, 1, 2, 3, 4]
```

2.4 Maps

Declaring an empty map

```
map x = map();
```

Declaring map with key-value pairs

```
map x = {"Michal":"Language Guru"};
```

Add a key/value pairing into the map:

```
map x = map();
x.put("Michal", "Language Guru");
x{"Michal", "Language Guru"}
```

Delete an key/value pairing from the map from the key:

```
map x = map();
x.put("Michal", "Language Guru");
x.put("Andrew", "Manager");
x.delete("Andrew")
x{"Michal", "Language Guru"}
```

Retrieve a value from the map from the key:

```
map x = map();
x.put("Michal", "Language Guru");
string a = x.get("Michal")
print(s);
"Language Guru"
```

Check if a map contains a specific key:

```
map x = map();
x.put("Michal", "Language Guru");
if(x.containsKey("Michal"))
{
     print("true");
}
true
```

Check if a map contains a value:

```
map x = map();
x.put("Michal", "Language Guru");
if(x.containsValue("Language Guru"))
{
     print("true");
```

```
    }
    True
```

Check if two maps are equal (contain the same data):

```
    map x = map();
    map y = map();
    x.put("Michal", "Language Guru");
    y.put("Michal", "Language Guru");
    if(x.isEqual(y))
    {
         print("true");
    }
    x.put("Emily", "Tester");
    if(!x.isEqual(y))
    {
         print("true");
    }
    True
    True
```

3. Operators

3.1 Arithmetic Operators

Type	Description
=	Assignment
+	Addition between ints, doubles, graphs; string concatenation; list appending
-	Subtraction between ints, doubles
*	Multiplication between ints, doubles
/	Division between ints, doubles
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%	Remainder for ints

3.2 Logical Operators

Type	Description
`==`	Compares two values and returns true if they are equal
`!=`	Compares two values and returns true if they are not equal
`<`	Returns true if the value on the left is less than the value on the right
`>`	Returns true if the value on the left is greater than the value on the right
`<=`	Returns true if the value on the left is less than or equal to the value on the right
`>=`	Returns true if the value on the left is greater than or equal to the value on the right
`&&`	Compares two boolean values and returns true if they are both true
`||`	Compares two boolean values and returns true if at least one is true

3.3 Graph Operators

Type	Description
`|`	Graph union
`&`	Graph intersection
`->`	Directed edge
`--`	Undirected/Bidirectional edge
`{}--`	Weighted edge (weight in brackets)
`~`	Delete node
`~>`	Delete edge

3.4 Further Examples: + Operator

Type	Description

`<list> + <*>`	Add element to the end of a list
`<string> + <string>`	Concatenate strings
`<graph> + <graph>`	Add all nodes and edges of one graph to another graph

3.5 Comments

Type	Description
`//`	Single line comments
`/*...*/`	Multiple-line comments

4. Graph Overview

4.1 Graph Declaration

Declare an empty graph and assign it to a graph variable `g`:

```
graph g = {{}};
```

Declare a graph with existing nodes `A`, `B`, and `C`:

```
graph g = {{A;B;C}};
```

Declare a graph with existing edges `E` and `F`:

```
graph g = {{E;F}};
```

To declare a graph with a list of nodes `m`:

```
graph g = {{m}};
```

Create a subgraph of graph g (data copied):

```
g{ A->B };
```

An anonymous graph is intuitively a graph not assigned to a variable:

```
{{ A->B }}
```

Since each node can only belong to one graph at a time (including anonymous graphs), a special double braces syntax is used in which addition, modification, or deletion of nodes and edges occurs:

```
g{{ }}
```

This syntax does not return the graph, or the subgraph with the specified nodes. It is also distinct from an anonymous graph due to the graph variable prefixing the braces. Examples of this syntax are shown in the below sections.

4.2 Nodes in Graphs

Declare a node outside of a graph (creates a node `A` with key: `"uni"` and data: `"mp3242"` and assigns it to variable `A)`:

```
	node A = {"uni" : "mp3242"};
```

Declare a node inside of a graph:

```
	graph g = {{ A={"uni" : "mp3242"}}};
```

Declare multiple nodes inside of a graph:

```
	graph g = {{ A={"uni" : "mp3242"; B={"uni" : "aq0000"} }};
```

Declare two new nodes and connect them with an edge inside of a graph:

```
	graph g = {{ A={"uni" : "mp3242"} -> B={"uni" : "aq0000"} }};
```

This creates a new node `A` with key: `"uni"` and data: `"mp3242"` and node `B` with key: `"uni"` and data: `"aq0000"` with a unidirectional edge from `A` to `B`.

Add node `A` to graph `g`.

```
	g{{ A }}
	g = g + {{A}}; //equivalent
```

If node `A` already exists in graph `g`, then `A` will just be updated.

Add nodes `A` and `B` to graph `g`.

```
	g{{ A; B }};
```

Delete a node or multiple nodes from graph `g`.

```
	g{{ ~A }};
	g{{ ~A; ~B }};
```

Modify a node's data outside of the graph.

```
	A.put("uni","aj0000");
```

Access node `A` from graph g and store in variable `N`.

```
	node N = g[A];
```

4.3 Edges in Graphs

To start, one should notice that there is no edge type in Graphiti, and there is no way for edges to be extracted from graphs or stored in variables. Nevertheless, it is simple to add, remove, and manipulate edges.

Declare graph g with unidirectional edge from node A to node B:
```
    graph g = {{ A -> B }};
```

Declare graph g with undirected/bidirectional edge from node A to B.
```
    graph g = {{ A -> B; B -> A }};
    graph g = {{ A -- B }}; //shortcut
```

An edge declared with `--` is a shortcut for declaring an edge in both directions with the same weight. In Graphiti, there are no true undirected edges; they are represented as bidirectional edges.

Declare a graph with multiple children. Parent node `A` with unidirectional edges to children nodes `B` and `C`:
```
    graph g = {{ A -> [B,C] }};
```

Declare a graph with parent nodes A and B and child node C with unidirectional edges from A to C and B to C.
```
    graph g = {{ [A,B] -> C }};
```

Declare multiple parents and children with edges between all nodes on the left to all nodes on the right of the `->` operator (i.e. `A->C, A->D, B->C, B->D`)
```
    graph g = {{ [A,B] -> [C,D] }};
```

Add an edge from node `A` to node `B` to graph `g`:
```
    g{{ A->B }};
```

If the edge from A to B already exists, then nothing changes. If either node (or both) has not been added to the graph, then it will be added automatically.

Declare edges between different nodes in a graph:
```
    g{{ A -> B -> C; C-> A }};
```

Delete an edge from node `A` to node `B` to graph `g`:
```
    g{{ A ~> B }};
```

Declare weight of `"employee_of"` on an edge from node A to node B in a graph.
```
    g{{ A {"employee_of"}-> B }};
```

Declare a non weighted edge:
```
    g{{ A {}-> B }};
    g{{ A -> B }}; //weight braces not necessary
```

All edges have a default weight of `null`.

Modify edge weight in graph g (the second option is a built-in function, see section 7):
```
g{{ A {"boss_of"}-> B }};
modify_edge(g, A, B, old_data, new_data);
```

Change edge direction:
```
g{{ A ~> B; B -> A}};
```

5. Keywords (Reserved Words)

In addition to the data types in section 2, there are a few more Graphiti keywords.

Type	Description
`all`	Used in a graph to get all nodes
`break`	Exit a loop immediately
`continue`	Skip to the next iteration of the loop immediately

6. Control Flow

6.1 Conditionals

If-else blocks follow a familiar C-style syntax:

```
if (condition) {
     statement;
} else if (condition) {
     statement;
} else {
     statement;
}
```

6.2 Loops

Graphiti uses C-style for- and while-loop syntax. Graphiti also offers a for-each loop to iterate through the nodes or edges of a given graph, as well as `break` and `continue` keywords.

```
for (int i = 0; i < n; i++) {
     statement;
```

```
      }

      while (true) {
            statement;
      }

      for (node n : g) {
            statement;
      }
```

7. Functions

Function declaration syntax is familiar as well:

```
      int add(int a, int b) {
            //logic
      }
```

Users must utilize a main function, where the program begins:

```
      int main() {
            //logic
      }
```

8. Built-In Graph Functions

8.1 Graph Methods

Function Header	**Function Purpose**
`graph add(graph A, graph B);`	Take graph A and graph B and copy all the nodes and edges. An example can be seen here: http://mathworld.wolfram.com/GraphUnion.html By definition, this graph would have to be an unconnected graph.
`graph union(graph A,  graph  B);`	Take graph A and graph B and filter out all the nodes where the data is congruent. As nodes support maps, all nodes with the same map (same keys AND values), delete that node and its respective edge.

`graph intersection(graph A, graph B);`	Take graph A and graph B and filter out all the nodes where the data is NOT congruent. As nodes support maps, all nodes with the same map (same keys AND values), delete that node and its respective edge.
`void print_graph(graph G);`	Print the Adjacency Matrix of the graph

8.2 Vertex and Edge Methods

Function Header	**Function Purpose**
`void modify_edge(graph G, node to, node from, string new);`	First it finds the edge that connects both nodes. Once the edge is found, it will delete the old weight and input its new weight
`void delete_edge(graph G, node to, node from, string relationship_name);`	First it finds the edge that connects both nodes. Once the edge is found, it will delete the old weight. If the edge has no weights, the edge structure will be deleted
`void modify_vertex(graph G, map old_data, map new_data);`	Use the old data to find the old node, once the node is found, replace the data with new data.

9. Sample Programs

```
/* Example 1: Hello World*/
int main()
{
	print("Hello World!");
	return 0;
}

/* Example 2: BFS*/
int main()
{
	//node declarations and graph creation omitted
```

```
        graph g; startnode n;
        map visited = map();
        for (node n : g) {
                visited.put(atoi(node.id), "false");
        }
        list queue = [];
        queue += startnode;

        while (queue.len() > 0) {
                node = queue.remove_head();
                print(node.data);
                for (n : {{node.children()}} ){
                        if (map.get(atoi(node.id)) == "false") {
                                map.put(atoi(node.id), "true");
                                queue += n;
                        }
                }
        }
        return 0;
}
```