Graphiti

Emily Hao	esh2160	Tester
Sydney Lee	stl2121	Tester
Michal Porubcin	mp3242	Language Guru
Andrew Quijano	afq2101	Manager
Alice Thum	at3061	System Architect

September 2018

1 Introduction

Graphiti is a language designed to simplify the process of representing graphs and expressing graph-related algorithms such as Dijkstra's shortest-path algorithm, breadth-first search, and topological sorting. Graphiti introduces methods that will allow programmers to easily create representations of graphs, manipulate edges and weights, and iterate through groups of nodes to simplify the implementation of these algorithms.

Additionally, with the rise of big data, there has been limited time to process it into SQL databases for analysis. SQL databases are not geared to mapping relationships between objects. Therefore, one NoSQL database solution was Neo4j which makes use of vertices, but also (directed/undirected) edges also contain the label of the relationship between the vertices. Finally, we intend to have our language compatible with Neo4j, such as being able to print the graph into a JSON format to be imported into a Neo4j database.

2 Language Overview

This graph requires various C structs to be implemented. The graph will contain both an adjacency matrix/vertex structs for easy access to the relevant data. The graph also records what kind of graph it is. In the case of unweighted graphs, all weights will be initialized to 0. With regards to undirected graphs, this would be represented as a two-way directed edge. There is also a linked list containing all the data for the edges.

3 Implementation Strategy

Will use our own C library. For the sake of keeping the graph data neat, we will have a separate linked list containing all the edges. For the vertices, the information will be stored in an adjacency linked list. An example of how this would look is shown in an image below.

4 Data Types

Type	Description/Notation
graph	set of nodes and a set of edges that connect a pair of nodes
int	4 bytes (32 bit) integer values
double	8 bytes (64 bits) decimal values
boolean	1 byte (8 bits) true/false
string	Finite sequence of characters
vertex	Struct containing id for the node and additional information
edge	Struct with the nodes that the edge connects and additional information
list	Linked list with unordered data. Can support any kind of data.

5 Operators and Syntax

Operator	Description
=	Assignment
+	Addition between ints, doubles, sets, graphs; string concatenation
-	Subtraction between ints, doubles, sets, graphs
*	Multiplication between ints, doubles
/	Division between ints, doubles
%	Remainder for ints, doubles
&	Intersection for graphs
—	Union for graphs
==	Logical is equal (doubles, integers, boolean, string, vertex, edge) Maybe check Graph isomorphism?
!=	Logical not equal to (doubles, integers)
<	Logical less than (doubles, integers)
>	Logical greater than (doubles, integers)
<=	Logical less than or equal (doubles, integers)
>=	Logical greater than or equal (doubles, integers)
&&	Conditional AND
|||	Conditional OR
;	Ends a statement, or ends graph subdefinition
–	Undirected edge
->	Directed edge
<>	Bidirectional edge
{}–	Weighted edge (shown for undirected edge)
//	Remove edge
~~	Flip direction of edge
:	Node initialization operator
,	Node separator

6 Basic Operations and Examples

6.1 Declare a graph

```
graph g = {A1,A0:1 -> B:2};
```

The colon operator indicates a new vertex. The data on the left of the colon is the variable name, while the element on the right of the colon is the vertex's data. We initialize two vertices to 1 with a comma.

6.2 Adding an element to existing graph

```
graph g = g{A -> C:null};
```

To reference vertices from previously declared graphs by variable name alone (without colon), prefix curly brackets with graph variable name. C does not exist yet, so data must be provided (even if it is null).

6.3 Getting a subset of a graph with vertex variable names

```
graph h = g{A,B};
graph h = g{1,2};
```

The graph will increment a counter to generate unique vertex identifiers. The second line builds graph h from the vertex ids of graph g.

6.4 Adding children

```
for (int i=0; i<5; i++)
{
    g = g{A->null:i};
}
```

This loop adds 5 anonymous children of A, with data i.

7 Control Flow

Graphiti will include if, else if, and else statements, for loops, and while loops all adhering to the syntax of C. In addition, an enhanced for loop will be provided in order to iterate easily through nodes or edges of a graph, with syntax as seen below.

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

8 Potential Standard Library Functions

Graph Functions

- Return all nodes in a graph
- Return all edges in a graph
- Find a node in a list
- Find neighbors of a node
- Find the distance between two nodes
- Add/remove node in a graph
- Add/remove edge in a graph

Neo4j Support

- Find all friends of Alice
- Find all second-degree friends of Jim

9 Potential Standard Library Function Implementation

```
//get all edges of a node
List get_all_edges(struct graph * G, struct vertex * current_vertex)
{
    // initialize list
    struct List list;
```

```
    initList(&list);

    // Search through the edges
    struct edge * edge= G -> edge_head;
    while(edge)
    {
        // If the edge is starting from the current node, add the edge to the list
        if(edge -> from->primary_key == current_vertex->primary_key)
        {
            addFront(&list, edge);
        }
        edge -> next_edge;
    }
    return list;
}

// get all friends of \Alice" in a graph

// input argument: name = \Alice", Relationship = \FRIEND_OF"
void get_friends(struct graph * G, String name, String relationship)
{
    int friends = 0;
    // find Node with right name
    struct vertex * current_vertex = G -> vertex_head;
    while(current_vertex)
    {
        if(current_vertex -> string_data == name)
        {
                break;
        }
        current_vertex = current_vertex -> next_vertex;
    }

    // Alice does NOT exist in this graph!
    if(current_vertex == NULL)
    {
        return -1;
    }
    else
    {
        struct List relation_list = get_all_nodes(current_vertex);
        // traverse List, check for correct relationship
        struct Node * current_node = relation_list -> head;
        while(current_node)
        {
            struct edge * current_edge = (struct edge *) current_node -> data;
            if(current_edge -> relationship_name == relationship)
            {
                print("A friend of Alice is: " + current_edge -> to -> string_data);
            }
        }
    }
}
```