yeezyGraph - a Graphing Language

Kiyun Kim (ckk2117)
Nancy Xu (nx2131)
Wanlin Xie (wx2161)
Yiming Sun (ys2832)

February 8, 2017

1 Introduction

yeezyGraph is a language that is built upon graph structures - a set of nodes and edges. Graph structures model the relationships (edges) between a group of objects (nodes). Because the structure of a graph itself is so simple, a graph can be used to represent much more complicated problems. For example, finite-state automata can be represented as directed graphs, puzzles can be reorganized as configuration graphs, and recursive algorithms can be shown as dependency graphs.

yeezyGraph is designed to facilitate the creation of graphs and the writing of graph algorithms, so that users are not bogged down by graph creation. yeezeGraph will not only implement the graph the user wishes to represent, but also output a model of the structure that is easily configurable for further applications. For the purposes of this language, all graphs are treated as directed graphs with weighted edges–that is, every edge travels in one direction, and every edge has a value assigned to it as its weight. Because undirected graphs and unweighted edges can be simulated using directed graphs and weighted edges, this choice has little impact on the user's ability to create a wide variety of graphs, and allows for a simpler syntax to support the goal of this language.

2 Language Features

2.1 Types and Literals

2.1.1 Primitive Types

Boolean (`bool`): the boolean data type has only two possible values - true and false.
Integer (`int`): an integer, typically reflecting the natural size of integers on the host machine.
Floating point (`float`): single-precision floating point.
String (`string`): a sequence of zero or more characters, enclosed in double quotes.

2.1.2 Derived Types

List (`map`): an ordered collection of elements.
Map (`map`): a mapping of keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
Queue (`queue`): holds elements prior to processing.
Node (`node`): a vertex of a graph. A node consists of the fields `value`, `visited`, `inNodes`, as well as `outNodes`. `value` is of type `string`, and represents a value that is associated with that particular vertex. `visited` is of type `bool`, and indicates whether the particular node has been visited or not.`inNodes` is of type `map`, which maps a `node` that has a directed edge into our node in question, to the value of the particular edge's edge weight. `outNodes` is of type `map`, which maps a `node` that has a directed edge out from our node in question, to the value of the particular edge's edge weight.
Graph (`graph`): a graph is a collection of nodes. A graph consists of the sole field `nodes`, which is a list of node vertices in the particular graph.

2.2 Operators

2.2.1 Variables and Assignment

The = operator is used to assign the value of an expression to an identifier.

2.2.2 Arithmetic Operators

The operators for integer and floating-point arithmetic are +, -, *, /, and %. The unary - operator has the highest precedence, followed by the binary *, /, and % operators, followed by the binary + and - operators. All arithmetic operators are left-associative. There is automatic promotion from `int` to `float`.

2.2.3 Logical and Relational Operators

The relational operators are <, <=, >, >=, which have the highest precedence, followed by the equality operators == and !=, then &&, then ||. When comparing equality, primitive types are compared by value, while derived and user-defined types are compared by reference.

2.2.4 String Operators

The + operator concatenates strings and returns a new string.

2.2.5 Node Operators

`->`: `node1 -> node2, num` adds a directed edge of weight `num` from `node1` to `node2`.
`!->`: `node1 !-> node2` removes the edge that exists between `node1` and `node2`.
`==`: `node1 == node2` compares node1 against node2 and returns true if they refer to the same node, false otherwise.

2.2.6 Graph Operators

No operators exist for the graph, as the graph is simply a collection of nodes.

2.3 Built-In Functions

2.3.1 Node Built-in Functions

`print(node)`: prints the name of the node.
`getNode(name)`: retrieves/creates a node with the given name.
`setNode(string name, List<Map<Node, weight>> incoming, List<Map<Node, weight>> outgoing)`: modifies the adjacency lists of the node with the given name.

2.3.2 Graph Built-in Functions

`addGraph(Graph 1, Graph 2, List<Node> map1, List<Node> map2, List<float> edgeWeights)`: returns a copy of the addition of the two graphs, whose connectivity is determined by the mapping of the nodes given by the two lists and the list of the edge weights.
`print(graph)`: outputs an image of the graph.
`iterateThrough()`: traverses the graph and prints out all the nodes within the graph.
`is_connected()`: returns a boolean value dictating the connectivity of the graph.
`has_unvisited_nodes()`: returns a boolean value stating whether the graph still has unvisited nodes.

3 Source Code

```
//Declaration of a node
    Node x1 = "1";
    Node x1 = "2";
    Node x1 = "3";
    Node x1 = "4";

//Adding edges
    x1->x2, 3
    x2->x3, 2
    x3->x4, 1
    x4->x1, 4

//Create a graph
    Graph g1;
    g1.add(x1);
```

```
    g1.add(x2);
    g1.add(x3);
    g1.add(x4);

//print graph
    print(g1)

//Implementation of a BFS
BFS (Graph G, Node s) {
    Queue q;
    while (Graph G.has_unvisited_nodes()) {
        q.enqueue s;
        s.visited = true;
            while (!q.empty()) {
                V = q.dequeue();
                print(V); // print processed node
                //process all neighbors of V
                for (int i = 0; i < V.inNodes.Count; i++) {
                    if (V.inNodes.getKey(i).visited = false) {
                        q.enqueue(V.inNodes.getKey(i))
                        V.inNodes.getKey(i)).visited = true
                }
            }
        }
    }
}
```