yeezyGraph - Language Reference Manual

Wanlin Xie (wx2161) - Manager
Yiming Sun (ys2832) - Language Guru
Nancy Xu (nx2131) - System Architect
Kiyun Kim (ckk2117) - Tester

February 22, 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. yeezyGraph 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 Lexical Conventions

2.1 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.

2.2 Derived Types

List (`list`): 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.

2.3 Comments

Comments are denoted by `**insert comment here**`. There is no difference between single and multi-line comments. Comments may not be nested.

2.4 Identifiers

Variable names must start with a lowercase letter. Node names are automatically generated as `graphname_nodename`.

3 Keywords

The following keywords are reserved words in yeezyGraph and cannot be used for variables or functions defined by the user: `graph, if, else if, else, for, while, bool, int, float, string, map, return`.

4 Built-in Data Types

4.1 node and graph objects

A `node` object represents a vertex in a `graph` object, and a `graph` object represents a collection of nodes.

Definition of graph objects:
- An empty graph is a graph
- A node cannot exist independently of a graph; a node is only created in association with a graph
- There can exist disconnected nodes within a graph object
- A node cannot belong to two distinct graphs
- A graph added to another graph is a graph
- A graph subtracted from another graph is a graph.

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. You cannot initialize a node; a node is only created when you add it to a graph.

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`. `Graphs` are immutable, meaning that you cannot modify its state after a `graph` is created. In order to make changes to an existing `graph`, you need to assign the modified graph to another `graph` instance.

4.2 Collections

List (`list`): a mutable, ordered set of elements.

Map (`map`): a set of `key`, `val` pairs used to represent edges between `nodes`. A `map` element consists of a `key` and `val` pair, where the `key` is the `value` of a particular `node` and the `val` is the weight of the edge between the two `node` objects. Each `node` should contain two `maps`, one to show incoming edges (`inNodes`) and one to show outgoing edges (`outNodes`).

Priority Queue (`pqueue`): a data structure where each element has an associated priority value. Elements with higher priorities are located further up in the `pqueue`.

5 Operators

5.1 Node Operators

{{}}: {`node name`} Accesses/creates a `node` object from the `node` name. The node is identified by node name and graph name. For example, if you want to create a `node` in `graph` g1, the name of the `node` would be g1_n1. You would only utilize this operator when adding/subtracting a `node` in a graph, or when accessing a field from a `node` object.
`@visited`: {{g1_n1}}`@visited` accesses the boolean visited field of `node` g1_n1.
`@inNodes`: {{g1_n1}}`@inNodes` accesses the inNodes field, which is a Map Collection, from `node` g1_n1.
`@outNodes`: {{g1_n1}}`@outNodes` accesses the outNodes field, which is a Map Collection, from `node` g1_n1.

5.2 Graph Operators

`g1::n1` adds a node of string value n1 to the graph g1. This is the only way to "construct" a node, which is uniquely identified by its graph owner and string value. Node n1's `visited` field is initialized to false upon construction, and its `inNodes` and `outNodes` maps are initialized to be empty. The :: operator can be chained.
`g1#n1` removes the node of string value n1 from the graph g1. Nodes adjacent to n1 will

have their `inNodes` and `outNodes` maps updated.
`g1 = {{g1_n1}} ->(x) {{g1_n2}}` adds or updates a directed edge of weight x from n1 to n2, returning a new graph object that is assigned to g1. The -¿ operator can be chained.
`g1 = {{g1_n1}} !-> {{g1_n2}}` removes the directed edge from n1 to n2, returning a new graph object that is assigned to g1. The `!->` operator can be chained.
`g1 == g2` compares the structural equality of graphs g1 and g2.

6 Source Code

```
**Declaring a graph**
graph g1;
graph g2;

**Adding nodes**
g1 = g1::n1;
g1 = g1::n2;
g1 = g1::n3;

g2 = g2::n1:n2:n3

**Adding connections between nodes**
{{g1_n1}}->(3){{g1_n2}}
{{g1_n1}}->(5){{g1_n3}}
{{g1_n2}}->(3){{g1_n3}}
{{g1_n2}}->(4){{g1_n1}}
```

7 Built-in Functions

7.1 Nodes

7.1.1 Node Built-in Functions

`printIncomingNodes(node name)`: retrieves the incoming node list and prints it.
`printOutgoingNodes(node name)`: retrieves the outgoing node list and prints it.
`hasConnections(node name)`: returns a boolean value indicating whether a node has any incoming or outgoing edges.
`returnGraphName(node name without graph name attached)`: returns a list of the graphs that contain a node with the node name, for example, printGraphName(n1).

7.2 Graphs

7.2.1 Graph Built-in Functions

`isEmpty(g1)`: checks if graph g1 is empty.
`size(g1)`: returns the number of nodes in graph g1.
`contains(g1, n1)`: returns true if the graph g1 contains the node n1; false otherwise.
`print(g1)`: prints to standard output the string representation of a graph.

7.3 Collections

7.3.1 List Built-in Functions

`print()`: prints the list of elements in order.
`get(int idx)`: returns the element at index `idx`. If index `idx` does not exist, an exception will be thrown.
`add(int idx, node)`: adds a `node` to the the `list` so that the new element is at index `idx`. If the list is not long enough to have index `idx`, then it will throw an exception.
`remove(int idx)`: removes the element at index `idx`. If the list does not contain that index, it will throw an exception.
`contains(string name)`: searches the list to see if it contains a node that has the name `name`. Returns true if it does, returns false otherwise.
`append(node)`: adds a node to the end of the `list`.
`removeAll()`:removes all elements from the `list`.
`isEmpty()`: checks if the `list`is empty. If it is empty, then it returns true. Otherwise, it returns false.
`size()`: returns the number of elements in the `list`.

7.3.2 Map Built-in Functions

`printMap()`: prints each key, val pair as a tuple.
`add(string key, double val)`: adds a `key, val` pair to the `map`.
`remove(string key)`: removes a `key, val` pair from the `map`.
`get(string key)`: returns the value associated with `key`. If `map` does not contain `key`, then an exception is thrown.
`contains(string key)`: checks whether or not the `map` contains `key`. If it does, then it returns true, otherwise it returns false.
`listKeys()`: generates and returns a list of all keys in `map`.
`size()`: returns the number of keys in `map`. `isEmpty()`:checks whether or not `map` contains any `key, val` pairs. If it contains none, then this returns true. Otherwise, it returns false.

7.3.3 Priority Queue Built-in Functions

`isEmpty()`: checks if the `pqueue` is empty. If it is empty, then this function returns true. Otherwise it returns false.
`size()`: returns the number of elements contained in the `pqueue`.
`top()`: returns the value of the element on the top of the `pqueue`, but does not remove it.
`push(node name)`: adds an element to the `pqueue` in accordance to the element's priority value.
`pop()`: removes and returns the element on the top of the `pqueue`.
`removeAll()`: removes all elements of the `pqueue`.

8 Control Flow

8.1 Logical Expressions

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.

8.2 If statements

```
if (condition) {

} else if (condition) {

} else {

}
```

8.3 For loops

```
for (node:graph) {

}
```

8.4 While loops

```
while (condition) {

}
```