Hippograph

A language for High Performance Parsing of Graphs

Benjamin Lewinter
Manager
bsl2121

Irina Mateescu
Language Guru
im2441

Harry Smith
System Architect
hs3061

Yasunari Watanabe
Tester
yw3239

September 19, 2018

1 Introduction

Hippograph will aim to be an improved version of *giraph*, a project from PLT Fall 2017, with improvements in graph creation and graph query capabilities. The language will support a range of graph types, like its predecessor; however, Hippograph's graph types will be inferred from arguments passed in during instantiation. Additionally we will use elements of the Cypher graph query language to ease graph search.

This language will be ideally suited for a wide range of graph problems that demand flexible representation of data and graph structures. Graphs can also be searched and traversed according to user-defined search strategy functions.

2 Data Types

graph	a set of nodes and edges
list	an ordered collection of data
set	an unordered collection of data
map	an unordered collection of key/value pairs
node	a container for a name-value pair
edge	a three-tuple of the source node, destination node, and weight/relationship
fun	a function, with input types, return type, and optional name
search	a strategy for searching/traversing graph, that takes in a node
int	a 4 byte integer
float	a 8 byte floating point
string	a sequence of characters
char	a 1 byte character
bool	a boolean true/false

3 Syntax

Hippograph inherits C-like syntax for basic operations, indentation, and line-delimitation conventions.

3.1 Functions

Hippograph allows the declaration and usage of both named functions and lambda functions.

```
fun return-type name(parameters) {statements} //named function signature
fun (parameters) : {statements} //lambda function signature
```

3.2 Control Flow

Hippograph supports `for_each` as well as graph-specific variations so as to make graph traversing easier.

```
for_each(int : range(int, int)) {statements}
for_node(node : graph, search) {statements}
for_edge(edge : graph, search) {statements}
```

The entry point in a Hippograph program is `main()`.

3.3 Graphs

When explicitly initializing a graph, we separate node and edge declarations to represent complex graphs more cleanly. We also define an alternative shorthand syntax for inlining node and edge declarations. A node is represented as a name-value pair, both primitive data types, with allowance for null values. An edge may be directed, and it may be weighted by data of a primitive type.

```
graph g = {
  nodes: {'a': 'h', 'b': 'e', 'c': 1, 'd': 'l', 'e': 0},
  edges: {'a' -- 'b' -- 'e', 'c' <-(10)- 'd', 'e' -(4.4)- 'd', 'b' --> 'c'}
}

graph h = {
    'a'{'h'} -- 'b'{1}, 'b' -(10)- 'c'{"Hello"}, 'd'{"Jane"} -("mother")-> 'f'{"
        Jen"}
}

g.add_edge(edge {'a' -(30)- 'd'})
h.add_edge(edge {'c' <-- 'b'})
```

This graph-related syntax can be summarized as follows:

Table 1: Graph Syntax

syntax	*description*
`--`	undirected edge, no specified weight
`-->`	singly directed edge, no specified weight
`<--`	singly directed edge, no specified weight
`-()-`	undirected edge, weight value specified between parentheses
`-()->`	singly directed edge, relation value specified between parentheses
`<-()-`	singly directed edge, relation value specified between parentheses
`:`	delimiter between key and value declaration for nodes
`[`	left boundary of graph lookup syntax
`]`	right boundary of graph lookup syntax
`{`	left boundary of graph data enumeration blocks
`}`	right boundary of graph data enumeration blocks

4 Standard Library

Table 2: Basic Operations

function	*description*
graph.nodes()	Get a list of all the nodes in the graph.
graph.add_node(*node*)	Add a node to a graph.
graph.remove_node(*node*)	Remove a node from a graph.
graph.has_node(*node*, *search*)	Return `true` if a graph has the given node using the search strategy.
graph.add_edge(*edge*)	Add an edge to a graph.
graph.remove_edge(*edge*)	Remove an edge from a graph.
graph.has_edge(*edge*, *search*)	Return `true` if a graph has the given edge using the search strategy.
graph.neighbors(*node*)	Get all neighboring nodes connected to the given node in a graph.
graph.print()	Print a visual representation of a graph.
graph.bfs(start, node_func, edge_func)	Traverse graph using bfs, calling the functions on each node or function traversed.
graph.dfs(start, node_func, edge_func)	Traverse graph using dfs, calling the functions on each node or function traversed.
edge.from(), *edge*.to()	Get the source or destination node connected by an edge.
edge.weight([*value]*)	Get the weight of an edge or set it with the provided value.
node.key([*value*])	Get the key of a node or set it with the provided value.
node.data([*value*])	Get the data of a node or set it with the provided value.
node.graph()	Get the graph that the node belongs to.

5 Sample Programs

5.1 Hello World

```
fun void main() {
    graph g = {
        nodes = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13}
        edges = {1 -('h')-> 2 -('l')-> 4 -(' ')-> 8,
                 4 -('w')-> 9, 2 -('l')-> 5 -('r')-> 11,
                 5 -('o')-> 10, 1 -('e')-> 3 -('o')-> 6 -('l')-> 12,
                 6 -('d')-> 13, 3 -(',')-> 7}
    }

    g.bfs(g[1], fun (node n) : {}, fun (edge e) : {print(e.weight())})

    return
}
```

5.2 Bellman-Ford Algorithm

```
fun void update(node n) {
    graph g = n.graph()
    for_node(next_n : g.neighbors(n)) {
        dist_src = next_n.data()
        updt_dist = n.data() + g[n, next_n].weight()
        next_n.data([min(dist_src, updt_dist)])
    }
    return
}

fun void main() {
    graph g = {
        nodes = {'S': 500, 'A': 500, 'B': 500, 'C': 500, 'D': 500, 'E': 500}
        edges = {'S' -(10)-> 'A' -(2)-> 'C' -(-2)-> 'B' -(1)-> 'A',
                 'S' -(8)-> 'E' -(1)-> 'D' -(-1)-> 'C',
                 'D' -(-4)-> 'A'}
    }

    list l = graph.nodes()
    for_each(no : range (1, l.len()-1) {
        g.bfs(g['S'], fun update (node n), fun (edge e) : {})
    }

    g.bfs(g['S'], fun (node n) : {print(n.key() + " : " + n.data() + "\n")}, fun
         (edge e) : {})

    return
}
```

6 Use Cases

This language can be used for a wide range of graph related programs, including:

- Shortest path searches
- Maximum flow
- Social networks/family trees

- Euler Paths / Hamiltonian cycles search