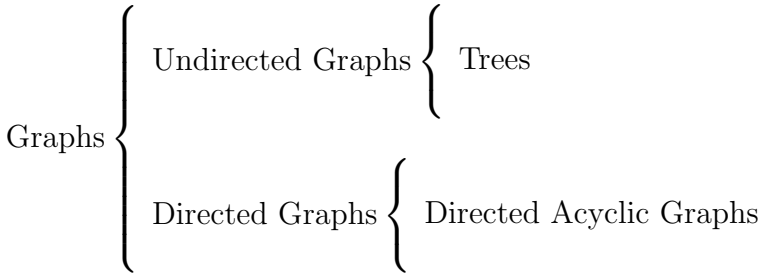
giraph

Daniel Benett (deb2174)
Seth Benjamin (sjb2190)
Jennifer Bi (jb3495)
Forrest Hofmann (fhh2112)
Jessie Liu (jll2219)

Fall 2017

1 Language Overview

We propose *giraph*, a language that simplifies graph implementation and manipulation. The language supports directed, undirected, and acyclic graphs. If time permits, we will extend support to bipartite, complete, and other properties. Users may initialize graphs by properties (i.e., a directed acyclic graph with 2 children per node – a binary tree) or manually, using intuitive node-edge syntax. Operators for join, union, intersection will make implementation of graph algorithms more concise and readable. The motivation for our language comes from previous experience with graph algorithms such as shortest path (Dijkstra's, Bellman-Ford, Floyd-Warshall), minimum spanning tree (Prim's/Kruskal's), and maximum flow (Ford-Fulkerson / Edmonds-Karp). Broader applications include network flow, vertex and edge coloring, and linguistic modeling.



2 Data Types

Type	Description
graph	a set of nodes and a set of edges
digraph	a graph with directed edges
tree	a graph that is undirected and acyclic
dag	an acyclic digraph with a source and a sink
node	a container for some built-in type
edge	a 2-tuple containing two nodes or a 3-tuple containing two nodes and a direction boolean
set	a finite unordered collection of unique built-in types
list	a finite ordered collection of built-in types
map	a collection of 2-tuples representing key-value pairs
int	a 4-byte integer
float	an 8-byte floating point value
string	a finite sequence of characters
character	a 1-byte value
boolean	a true or false value

3 Syntax

3.1 General Syntax

Syntax Literal	Description
{	a block indicator
}	a block terminator
(	an expression indicator
)	an expression terminator
,	a value separator
;	a statement terminator
//	a single line comment indicator
/*	a multiple line comment indicator
*/	a multiple line comment terminator

3.2 Reserved Keywords

Keyword	Description
`if (`*`expression`*`)`	a control flow indicator that executes the following block if *expression* is true
`elif (`*`expression`*`)`	a control flow indicator that executes the following block if previous if and elif expressions were not true and *expression* is true
`else`	a control flow indicator that executes the following block if previous if and elif expressions were not true
`for (`*`statement 1`*`; `*`expression`*`; `*`statement 2`*`)`	a loop that performs *statement 1*, executes while *expression* is true, and performs *statement 2* after every iteration
`for (`*`type iterating_var`*`; `*`collection`*`)`	a loop that executes once for every element in *collection*, with the element currently iterated on accessible with *iterating_var*
`while (`*`expression`*`)`	a loop that executes while *expression* is true
`function`	a function indicator
`lambda`	an anonymous function indicator
`void`	a void return type indicator
`return`	a return indicator

3.3 Operators

Operator	Description
`=`	assignment operator
`+`	addition operator valid for int, float, string
`-`	subtraction operator valid for int, float, string
`*`	multiplication operator valid for int, float
`/`	division operator valid for int, float
`%`	remainder operator valid for int, float
`&`	intersection operator valid for graph, set
`|`	union operator valid for graph, set
`&&`	conditional and operator valid for boolean
`||`	conditional or operator valid for boolean
`!`	logical complement operator valid for graph, boolean
`==`	relational equal to operator valid for all types
`!=`	relational not equal to operator valid for all types
`<`	relational less than operator valid for int, float
`<=`	relational less than or equal to operator valid for int, float
`>`	relational greater than operator valid for int, float
`>=`	relational greater than or equal to operator valid for int, float

3.4 Graph Syntax

Syntax Literal	Description
`--`	an undirected edge
`->`	a singly-directed edge
`<->`	a doubly-directed edge
`:`	a node initialization indicator
`[`	a node access indicator
`]`	a node access terminator

4 Standard Library

- Accessors/iterators:
  - Iterators/accessors for graphs: `nodes()`, `edges()`, `find(data)`, `connected_components()`
  - Iterators/accessors for trees: `root()`, `leaves()`
  - Accessors for DAGs: `source()`, `sink()`
  - Accessors for edges: `.from`, `.to`, `.weight`
  - Accessors for nodes: `.name`, `.data`
- Mutators: `add_node()`, `add_edge()`, `remove_node()`, `remove_edge()`, `Graph g1 + Graph g2`, `g1 - g2`
- Traversal: `bfs(graph g, node r, lambda f)`, `dfs(graph g, node r, lambda f)`, `preorder(tree t, lambda f)`, `inorder(tree t, lambda f)`, `postorder(tree t, lambda f)`
- Visualization: `render(graph, filename)`

5 Typical Use Cases

1. Shortest paths (Dijkstra's, Bellman-Ford, Floyd-Warshall)
2. Minimum spanning tree (Prim's/Kruskal's)
3. Maximum flow and minimum cut (Ford-Fulkerson/Edmonds-Karp)
4. Finding Eulerian tours and Hamiltonian cycles

6 Example Programs

1. Hello World

```
function void hello_world() {
    dag hello_world = A:'h' -> B:'e' -> C:'l'
                    -> D:'l' -> E:'o' -> F:' '
                    -> G:'w' -> H:'o' -> I:'r'
                    -> J:'l' -> K:'d';
```

```
    bfs(hello_world, A, lambda (node n) { print(n.data); });
}
```

2. Edmonds-Karp Algorithm

```
function dag augment(dag flow, dag path) {
    // Get bottleneck capacity of path.
    int min = path.edges()[0].weight;
    for (edge e: path.edges()) {
        if (min > e.weight) {
            min = e.weight;
        }
    }

    // Augment flow.
    for(edge e : path.edges()) {
        if (flow.has_edge(e.from, e.to)) {
            int current_flow = flow.get_edge(e.from, e.to).weight;
            // Add bottleneck capacity to current flow.
            flow.add_edge(e.from, e.to, current_flow + min);
        } else {
            int current_flow = flow.get_edge(e.to, e.from).weight;
            // Subtract bottleneck capacity from current flow.
            flow.add_edge(e.to, e.from, current_flow - min);
        }
    }
    return flow;
}

function digraph make_residual_graph(dag flow, dag network) {
    digraph residual_graph;
    for (edge e : flow.edges()) {
        int forward = network.get_edge(e.from, e.to).weight - e.weight;
        int backward = e.weight;
        if (forward > 0) {
            residual_graph.add_edge(e.from, e.to, forward);
        }
        if (backward > 0) {
            residual_graph.add_edge(e.from, e.to, backward);
        }
    }
    return residual_graph;
}

function dag edmonds_karp(dag network) {
    // The argument "network" contains the capacities as weights on edges.
    // Flow is represented with a graph exactly equivalent to network, but
    // with the flow on each edge as the weight instead of the capacity.
    // First, set up initial flow of 0 on every edge.

    dag flow;
```

```
    for (edge e : network.edges()) {
        flow.add_edge(e.from, e.to, 0);
    }
    dag residual = make_residual_graph(flow, network);
    while (true) {
        map parents;
        // Find shortest s-t path with BFS.
        bfs(residual, network.source,
            lambda(node n) {
                for (node neighbor : residual.get_neighbors(n)) {
                    parents[neighbor] = n;
                }
            });
        // If we didn't reach the sink, there is no s-t path in residual
        if (!parents.contains_key(i)) {
            break;
        }
        dag path;
        node i = network.sink;
        while (i != network.source) {
            path.add_edge(parents[i], i,
                          residual.get_edge(parents[i], i).weight);
            i = parents[i];
        }
        flow = augment(flow, path);
    }
    return flow;
}
```