giraph LRM

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

October 17, 2017

1 Introduction

Graph algorithms are relevant in subject areas like economics and chemistry, and have applications ranging from airline scheduling to linguistic modeling. Graphs are a fundamental method of representing the world we live in. Graph creation and algorithms in common programming languages can be tedious to write.

giraph is a language that focuses on simplifying the construction and manipulation of graphs and performance of graph algorithms. This happens through an intuitive, readable syntax for graph construction, graph types that allow the programmer to put more focus on writing algorithms instead of writing exceptions, and built-in functions like breadth first search, depth first search, and tree traversals.

2 Data Types

giraph is a statically typed language, and supports the following:

2.1 Primitive types

- `bool` - Boolean data, which can be `true` or `false`.
- `int` - Ints are signed 8-byte literals. Represents a number as a sequence of digits.
- `float` - Floats are signed double-precision floating point numbers.

2.2 Reference types

- `graph` - A graph consists of nodes and optionally edges. All nodes in a graph must have data of the same type. This type is designated when a new graph is declared with syntax: `graph<type> new_graph;`

  - *graph*`.add_node(`*node*`)` - add *node*
  - *graph*`.add_edge(`*edge* - add *edge* to *graph*
  - *graph*`.remove_node(`*node*`)` - remove *node* from *graph*
  - *graph*`.remove_edge(`*edge*`)` - remove *edge* from *graph*
  - *graph*`.has_node(`*fnode*`)` - returns *true* if *graph* has node *node*, false otherwise
  - *graph*`.has_edge(`*from_node*`,` *to_node*`)` - returns *true* if *graph* has an edge from *from_node* to *to_node*, false otherwise
  - *graph*`.find(`*data*`)` - returns first node in *graph* found containing *data*
  - *graph*`.neighbors(`*node*`)` - returns graph containing all neighbors of *node* as nodes

- `edge` - An edge connects two nodes within a graph, and can be directed or undirected, depending on whether it is in a graph or a digraph.
  - *edge*`.from()` - gives node *edge* is coming from. In an undirected graph, *edge*.from() != *edge*.to(), except in a self-loop. Beyond that, there is no guarantee as to which node will be returned by *edge*.from()
  - *edge*`.to()` - gives node *edge* is going to. In an undirected graph, *edge*.from() != *edge*.to(), except in a self-loop. Beyond that, there is no guarantee as to which node will be returned by *edge*.to()

- `wedge` - A weighted edge connects two nodes within a weighted graph and has an associated numeric weight.
  - *wedge*`.from()` - gives node *wedge* is coming from, same guarantees as *edge*.from()
  - *wedge*`.to()` - gives node *wedge* is going to, same guarantees as *edge*.to()
  - *wedge*`.weight()` - gives weight of *wedge*

- `node` - Nodes hold data within a graph. This data can be of any type. The data type is specified at node creation with syntax: `node<type> new_node;`
  - *node*`.data()` - gives data stored in *node*

- `string` - A sequence of ASCII characters. Literals are enclosed in double quotes, as so: `string s = "graphs are cool"`

2.3 Graph subtypes

The following distinct subtypes of `graph` are supported, with the following hierarchical structure:

- `tree` - An undirected graph that must be acyclic
  - *tree*`.root()` - gives root node of *tree*
  - *tree*`.leaves()` - gives a graph with all leaves *tree* and no edges
- `digraph` - A graph whose edges are all directed
  - `dag` - A digraph that must be acyclic
    - *dag*`.source()` - gives source node of *dag*
    - *dag*`.sink()` - gives sink node of *dag*
- `wegraph` - A graph whose edges are all weighted
  - `wedigraph` - A digraph whose edges are all weighted
    - `wedag` - A wedigraph that must be acyclic (aka: a weighted dag)
      - *wedag*`.source()` - gives source node of *wedag*
      - *wedag*`.sink()` - gives sink node of *wedag*

3 Operators and Expressions

3.1 Variable Assignment

Variables are assigned using the = operator. The left hand side must be an identifier while the right hand side must be a value or another identifier. The LHS and RHS must have the same type, as conversions or promotions are not supported. The variable assignment operator groups right-to-left.

3.2 Node Data Assignment

Nodes are assigned using the : operator. The node assignment operator has higher precedence than all other graph operators, so node data assignment and graph assignment can be accomplished in one line. Example, graph and node declaration and assignment:

```
digraph<int> g = A:1 -> B:3;
```

3.3 Arithmetic Operators

Arithmetic operator precedence is as in standard PEMDAS. Operator binding is as follows:

Additive operators $+, -$ group left-to-right.
Multiplicative operators $*, \backslash, \%$ group left-to-right.
Parentheses have the highest precedence, and therefore can be used to override the default operator precedence.

3.4 Logical and Relational Operators

Relational operators `(<, >, <= , >=)` and logical operators `(&&, ||)` also group left-to-right. So, a statement like `a < b && b < c && c < d` can simply be accomplished with `a < b < c < d`. Expressions with relational and or logical operators return 0 or 1 for true or false respectively.

Equality operators also group left-to-right, but have lower precedence than relational ones. Logical operators have the lowest precedence of the three.

3.5 Graph Construction

Graph construction uses intuitive notation, establishing nodes and including edges (denoted by `-`) between connected nodes. Example, a simple undirected graph:

```
graph<int> my_graph = A - B - C - D;
```

Directed graph construction uses intuitive arrow notation. Right arrows are left associative and left arrows are right associative. Doubly-directed edges and undirected edges are left associative. Example, a cyclic digraph:

```
digraph<int> my_graph = A -> B -> C -> E -> D -> C;
```

Parentheses can be used to nest declarations. Nodes and edges are still created left-to-right. Example, a binary tree with nested declaration:

```
tree<int> my_graph = A - (B - (D, E), C - (F, G));
```

Weighted graphs can also be constructed in this way. In this case, the edge notation is broken up with weights enclosed in square brackets. Example, an weighted undirected graph:

```
wegraph<int> my_graph = A -[3]- B -[4]- C -[-5]- D;
```

Example, a weighted digraph:

```
wedigraph<int> my_graph = A -[3]-> B <-[4]-> C <-[-5]- D;
```

In this last example, B and C both have directed edges pointing to each other with weight 4. If a wedigraph contains two nodes with outgoing edges to one another with different weights, those edges cannot be denoted using the doubly-directed edge operator.

3.6 Graph Operators

Graph operators are overloaded arithmetic operators, to allow for concise graph manipulation, for example, graph union and graph intersection. In some cases graph operations can simply be considered set operations. Operators `&, |, \` are binary operators that return a copy of the set union, set intersection, set difference of the original graphs respectively. A graph operator used multiple times groups left-to-right, although it does not matter. Combining different graph operators in one statement requires parentheses to specify precedence, otherwise the implementation is free to evaluate the expression in any order.

3.7 Comments

Comments in *giraph* will be written as such: `!~this is a comment~!`. Comments can be single or multi-line.

4 Control Flow

4.1 Conditionals

If-else statements are allowed, in the following formats:

```
if (condition) {statements}
if (condition) {statements} else {statements}
if (condition) {statements} else if {statements} else {statements}
```

The else block is optional for any if statement, and any number of else if's can be appended to any if statement.

4.2 Loops

C-style while loops and for loops are provided, such as the following:

```
while (condition) {statements}
for (initialization; condition; update) {statements}
```

They can either be followed by a single statement to be looped, or by a sequence of statements enclosed within brackets. Graph-specific iteration over nodes and edges is also allowed, using "for each" loops, which take the following format:

```
for_node(node : graph) {statements}
for_edge(edge : graph) {statements}
```

These iterate over all the nodes/edges of *graph* respectively, executing the looped statements at every node/edge. For example, the following loop can be used to print the data at every node in some graph g:

```
for_node(n :  g) print(n.data())
```

Each of `for_node` and `for_edge` iterate over their respective graph components in an unspecified order. However, one can also iterate over graph components in a specific order using the following loop constructions:

```
bfs(node :  graph ; root) {statements}
dfs(node :  graph ; root) {statements}
```

These iterate over the nodes of a graph using breadth-first search and depth-first search respectively, starting at *root* and executing the looped statements at every subsequently reached node. Thus, for some DAG g, one could write the following to print the data in all nodes in BFS order, starting with the source:

```
bfs(n :  g ; g.source()) print(n.data())
```

For trees, pre-order, in-order and post-order traversal are also provided:

```
preorder(node :  tree) {statements}
postorder(node :  tree) {statements}
inorder(node :  tree) {statements}
```

5 Program Structure

Programs in *giraph* consist of a sequence of functions, including a `main()` function which is the entry point of a compiled executable *giraph* program. Functions are defined with the following signatures:

```
return_type function_name(type arg, type arg, ...)  {body}
```

The `main()` method has return type void, and the program exits upon its completion.

6 Standard Library

6.1 Lists

Lists are stored internally as trees where every node has exactly one child, except the tail of the list which has no children. The following library functions are provided for list operations, provided that the tree's list structure is not tampered with outside of these functions:

- `List.new_list()` - returns an empty list
- `List.is_empty(`*list*`)` - returns *true* if *list* is empty, otherwise returns $false$
- `List.add_front(`*list*`, `*item*`)` - adds *item* at start of *list*
- `List.add(`*list*`, `*item*`)` - adds *item* at end of *list*
- `List.add(`*list*`, `*index*`, `*item*`)` - adds *item* at *index* in *list*
- `List.remove(`*list*`, `*index*`)` - remove item at *index* from *list*
- `List.get(`*list*`, `*index*`)` - gets item at *index* in *list*
- `List.size(`*list*`)` - gives number of items in *list*
- `List.append(`*list*1`, `*list*2`)` - All elements of *list*2 are added, in order, after all the elements of *list*1

To iterate over all elements in a list in order, one can do preorder traversal over the internal tree, as follows:
`preorder(`*node* ` : ` *list*`) {statements}`

6.2 Sets

Sets are stored internally as trees. The following library functions are provided for set operations, provided that the tree's structure is not tampered with outside of these functions:

- `Set.new_set()` - returns an empty set
- `Set.add(`*set*`, `*item*`)` - add *item* to *set*
- `Set.add_all(`*set*`, `*graph*`)` - Add all data in nodes in *graph* to *set*
- `Set.remove(`*set*`, `*item*`)` - remove node with data *item* from set
- `Set.contains(`*set*`, `*item*`)` - returns *true* if *set* contains *item*, or $false$ if does not
- `Set.is_empty(`*set*`)` - returns *true* if *set* is empty, otherwise returns $false$
- `Set.size(`*set*`)` - gives number of elements in *set*
- `Set.is_subset(`*set*`, `*subset*`)` - returns *true* if *subset* is a subset of *set*
- `Set.union(`*set*1`, `*set*2`)` - returns union of *set*1 and *set*2
- `Set.intersection(`*set*1`, `*set*2`)` - returns intersection of *set*1 and *set*2

6.3 Maps

Maps are stored internally as trees, similarly to sets. The following library functions are provided for map operations, provided that the map's structure is not tampered with outside of these functions:

- `Map.new_map()` - returns an empty map
- `Map.add(`*map*`, `*key*`, `*value*`)` - add *key/value* pair to *map*
- `Map.remove(`*map*`, `*key*`)` - remove *key* from map
- `Map.get(`*map*`, `*key*`)` - returns value in *map* corresponding to *key*
- `Map.contains(`*map*`, `*key*`)` - returns *true* if *map* contains a key-value pair keyed on *key*, or $false$ if does not
- `Maps.is_empty(`*map*`)` - returns *true* if map is emoty, otherwise returns $false$
- `Maps.merge(`*map*1`, `*map*2`)` - returns a map with all keys from *map*1 and *map*2. When both maps have a key, the key/value pairs of *map*1 take precedence. Neither *map*1 nor *map*2 are affected by this operation.