GRAIL: A Graph-Construction Language

Aashima Arora (aa3917), Rose Sloan (rns2144),
Jiaxin Su (js4722), and Riva Tropp (rtt2114)

February 8, 2017

1 Introduction

GRAIL (Graph Rendering Articulate Innovation Language) offers an innovative way to construct and manipulate graphs. The language provides built-in data structures for nodes, edges, and graphs. Users can construct directed or undirected graphs by adding nodes, edges and more. Furthermore, GRAIL empowers users with various operators – edge operators, list operators, and graph operators – to easily manipulate graphs. The goal of the language is to make graph construction and manipulation easier as well as allow for users to build complicated graphs through mathematical functions and simple objects, providing a powerful tool for graph applications in mathematics and computer science.

2 Programs in GRAIL

Graphs are a powerful method of representing and visually organizing data, but very few languages provide a robust inbuilt framework for solving graph problems. To prevent programmers from getting bogged down by the intrinsic details of the implementation of the graph algorithms and help them focus more on the problem at hand, GRAIL provides a much needed framework for handling graphs. These graphs can be used to model a number of mathematical and real world problems, including social network graphs, transportation networks, utility graphs, document link graphs, packet flow, neural networks, dependency modeling, and much more. In our language, nodes and edges will function as primitive data types and we will allow for sufficient flexibility to create undirected, directed and bidirectional graphs. The standard graph algorithms like path-finding and shortest path algorithms will be implemented as a part of the standard library.

3 Parts of GRAIL

Primitives:

- boolean
- char
- double
- int
- void

Objects:

- graph: a collection of undirected edges connecting primitives of a stated type
- digraph: a graph with directed edges
- edge: connects two primitive nodes, which can be directed or undirected
- list: an ordered array containing any number of objects of the same data type. Declared as type[].
- string: a character array

Integer and Double Operators:

`+, -, /, *, %, +=, -=, /=, *=, <, >, <=, >=, ==, !=`

Logical Operators:

`&&, ||, !`

Control Flow:

/* */	Comment
//	Single line comment
;	Signifies the end of a statement
if(...){} [else if(...){}] [else{}]	Conditional statements
for(...){} while(...){}	Loops

int myfunc(int x){
  return x;
}
	Functions

Graph Operators:

graph1 + graph2	Returns a graph containing all the nodes and edges in both graph1 and graph2
graph1 - graph2	Returns a graph containing the nodes of graph1 and all the edges in graph1 that do not appear in graph2
graph1[x, y : conditionals]	Returns a graph containing the nodes of graph1 and all the edges whose endpoints satisfy the conditional

List Operators:

graph*	Returns the list of nodes in the graph
list + item	Adds the item as the last element of list, if they are of the same type
list[x : conditionals]	Returns a list containing only the elements that satisfy the conditional

Edge Operators:

$a- > b$[(w)] $b < -a$ [(w)] $a - -b$ [(w)]	Returns an edge (when use on primitives or strings) or graph of edges (when used on lists) in the specified direction. The objects a and b must be primitives of the same type, both strings, or lists of the same length containing the same data type. The three operators are functionally equivalent in a graph, and in a digraph, the $--$ operator returns two edges between a and b, one in each direction. (w) is an optional argument and denotes the weight of the edge. The default value when not provided is 1.
.weight	The weight of the specified edge. Can be used to access or update the weight.
.to	The first node connected to the specified edge (source node in a digraph). Can be used to access or change the endpoint.
.from	The second node connected to the specified edge (destination node in a digraph). Can be used to access or change the endpoint.

Functions:

graph.display()	Displays a visual representation of the graph
graph.sort(feature, [direction])	Sorts the edges in a graph by the designated feature (the weight, the originating node, or the destination node). Direction is optional and can be denotes by the keywords "asc" or "desc" for ascending and descending. If no direction is specified, ascending is the default.
size(obj)	Takes in either a graph or list. Returns the number of edges in a graph or number of elements in a list
to(node, [s, l])	Returns a list of edges going to the node. If provided the optional second argument, returns only a single edge going to the node, the shortest if s is specified and the longest if l is.
from(node, [s,l])	Returns a list of edges from the node or a single edge if the second argument is provided, similar to how to works.
print()	Can be used to print strings, edges, ints, or lists

Other:

- Grail.Math.INF: the largest supported integer value

4 A Sample Program

The following program implements Djikstra's Algorithm for finding a shortest path.

```
int[] getclosestpaths(graph g, int s){
    g.sort(weight, asc);

    int[] dist = [size(g)]; //initializes an array of ints of size size(g)
    int[] prev= [size(g)];
    boolean[] visited = [size(g)];

    int inf = Grail.Math.INF;
    for(int i = 0; i < size(dist); i++){
        dist[i] = inf;
        visited[i] = false;
    }
    dist[s] = 0;

    for(int i = 0; i < size(g); i++){
        next = closestNode(dist, visited);
        visited[next] = true;

        int[] neighbors = g.from(next, s);
```

```
        for(int j = 0; j < size(neighbors); j++){
            int n = neighbors[j];
            int d = neighbors[j].weight + dist[next];
            if(dist[n] > d){
                dist[n] = d;
                prev[n] = next;
            }
        }
    }
    return pred;
}

int closestNode(int[] dist, boolean[] visited){
    int d = Grail.Math.INF;
    int n;
    for(int i = 0; i < size(dist); i++){
        if(v[i] == false && dist[i] < d){
            n = i;
            d = dist[i];
        }
    }
    return n;
}

//initialize a graph, add a node and an edge, and run closest paths algorithm
int graph g = {1->2,2->3,3->4};
g* += 5;
g += 4->5;
int[] allPaths = getClosestPaths(g,1);
```