Graphing Ain't Easy (GaE)

Kevin Zeng (UNI: ksz2109)
Andrew Jones (UNI: adj2129)
Jason Delancey (UNI: jrd2172)

Introduction and Motivation

Graphs are prevalent in many different places across the internet. Every social media platform has graphs representing friends (e.g. Facebook) or followers (e.g. Twitter). The goal of this language would be to allow the programmer to create a language that provides a much easier way to create and manipulate graphs with any data type stored inside the nodes, including custom structs. With easy to use syntax, Dijkstra's algorithm, Kruskal's algorithm, and other commonly used algorithms should be easily implemented. Any program that relies on a graph data structure to hold and manipulate data should be ideal for this language.

Description

This is a language used to create graphs for analytics, reporting, computation, and any other graph-intensive application. The syntax and features of the language are influenced by C and Golang. The idea is to create a user-friendly syntax for those who want to project their data on different types of graphs. The source of the data could be created from the language library or imported via file I/O (e.g. reading from a CSV file) and then manipulated using the language library. The language will use basic primitive data types for classifying objects. An example of such data types are Integers, Doubles, Booleans, Characters. It will also support other data types like Strings, Arrays, Maps, Graphs, and Structs.

Syntax

Data Types:

`int`	32 bit signed integer
`double`	32 bit floating point number
`bool`	Boolean - 0 is false, 1 is true
`char`	ASCII character
`string`	An array of ASCII characters
`array`	Fixed length list that can store any other type
`map<k,v>`	Variable-size mapping that associates key of type k to value of type v
`graph<n,e>`	Weighted and directed graph with nodes of type n and edges of type e
`struct`	A group of data elements grouped together under one name as a type definition

Operators:

`+, -, /, *, ++`	Integer operations
`+. , -. , *. , /.`	Double operations
`:=`	Variable declaration and initialization
`=`	Reassignment
`==, <, >, <=, >=`	Value comparison, used for int, double, bool, char
`===`	Deep comparison, used for array, map, graph, and struct
`||, &&, !`	Or, and, not (boolean logic operators)

Control Flow:

`//`	Single line comment
`/* … */`	Multiple line comments
`if, else if, else`	Conditional statements
`for, while`	For-loops and while-loops
`func`	Declaring a function
`;`	End of statement

Array Built-in Functions:

`len(array)`	Returns the length of a given array
`array[i]`	Returns the ith element in the array

Map Built-in Functions:

`len(map)`	Returns total number of key-value pairs in the map
`map[k]`	Returns value mapped to given key k. If key k doesn't exist in map, throw an error.

Graph Built-in Functions:

Note: g is a graph variable

`g.addNode(value)`	Adds a node to the graph, if a node with same value exists, return false, else returns true
`g.nodes()`	Returns an array of the values of the nodes
`g.addEdge(src, dest, edge)`	Adds edge to a graph, if an edge with the same source and destination exists, return false, else return true. Note that type of src and dest must be the same as type of node specified by the graph, and type of edge specified must be same type of edge specified by the graph.
`g.edges()`	Returns an array of the values of the edges
`g.getEdges(node)`	Returns edges starting from node (node is described with its value, e.g. g.getEdges(1) would get the edges from the node with value 1)

`g.getDest(node, edge)`	Returns the destination node of the edge that starts at the given node

Other Built-in Functions:

`print(value)`	Print the contents of 'value' to standard output
`print()`	Print the contents of the implicit parameter (e.g. g.print() where g is a graph)

Key Words:

`return`	terminates the function and returns the value
`continue`	Goes directly to the next iteration of the parent loop
`in`	Returns whether the left argument is contained by the right argument (bool)
`typedef`	used for declaring a structure
`range`	Used for iterating through a collection (map or array)

Example Source Code

```
typedef struct Int {
  value: int,
};

func get_fc_undirected_graph(int node_count, int value) graph<Int, Int> {
  graph<Int, Int> g := {};
  for int i := 0; i < node_count; i++ {
    g.addNode({value: i});
  }
  for int i := 0; i < node_count; i++ {
    for int j := 0; j < node_count; j++ {
      if i == j {
        continue;
      }
      g.addEdge({value: i}, {value: j}, {value: value});
    }
  }
```

```
    return g;
  }

  // returns total path weight of shortest path
  func get_shortest_path_length(graph<Int, Int> g, Int s, Int d) int {
    if !(s in g || d in g) {
      return -1;
    }
    // dijkstra's
    Int[] nodes := g.nodes;
    int node_count := len(nodes);
    map<Int, bool> visited;
    map<Int, int> dist;
    for int i := 0; i < node_count; i++ {
      dist[nodes[i]] = 100000000;
      visited[nodes[i]] = false;
    }
    dist[s] = 0;
    visited[s] = true;

    // instead of using a priority queue, we get the closest node on each loop
    for int i := 0; i < node_count; i++ {
      Int node := get_closest_node(visited, dist);
      visited[node] = true;

      Int[] edges := g.getEdges(node);
      for int j := 0; j < len(edges); j++ {
        int cur_best := dist[g.getDest(s, edges[j])];
        int next_best := dist[s] + edges[j].value;
        if next_best < cur_best {
          dist[g.getDest(s, edges[j])] = next_best;
        }
      }
    }

    return dist[d];
  }

  func get_closest_node(map<Int, bool> visited, map<Int, int> dist) Int {
    int lowest_so_far := 1000000;
    Int closest_node := null;
    for node, value := range dist {
      if dist[node] < lowest_so_far && !visited[node] {
        lowest_so_far = dist[node];
        closest_node = node;
      }
    }
    return closest_node;
  }
```

```
67
68    func main() {
69      graph<Int, Int> g := {
70        ({value: 1},{value: 2},{value: 1}), // syntax: (src, dest, edge)
71        ({value: 1},{value: 3},{value: 1}),
72        ({value: 1},{value: 4},{value: 1}),
73        ({value: 2},{value: 3},{value: 1}),
74        ({value: 2},{value: 4},{value: 1}),
75        ({value: 3},{value: 4},{value: 1}),
76        ({value: 2},{value: 1},{value: 1}),
77        ({value: 3},{value: 1},{value: 1}),
78        ({value: 4},{value: 1},{value: 1}),
79        ({value: 3},{value: 2},{value: 1}),
80        ({value: 4},{value: 2},{value: 1}),
81        ({value: 4},{value: 3},{value: 1})
82      };
83      print(g === get_fc_undirected_graph(4,1)); // outputs true
84      g = {
85        ({value: 1},{value: 2},{value: 1}), // syntax: (src, dest, edge)
86        ({value: 1},{value: 3},{value: 1}),
87        ({value: 1},{value: 4},{value: 1}),
88        ({value: 2},{value: 3},{value: 3}),
89        ({value: 2},{value: 4},{value: 5}),
90        ({value: 3},{value: 4},{value: 1})
91      };
92      print(get_shortest_path_length(g, 1, 4)); // outputs 1
93    }
94
```