Graphene

Ashar Nadeem	an3056
Shengtan Mao	sm4954
Vasileios Kopanas	vk2398
Matthew Sanchez	mcs2307

1 Introduction

Graphs have become one of the most common ways to store and represent data and information in Computer Science, being used in nearly every application from social media platforms to mapping services such as Waze. The norm with these graphs, and their algorithms, is needing to deal with messy implementations with lots of code and moving pieces. There is an unequivocal need to create a language which accurately allows programmers to create, represent and manipulate graphs while abstracting and simplifying these functions. The motivation for our language arose from the complex nature of graphs employed in representing and resolving the most intricate problems. The purpose of our language is to create graphs and implement commonly used graph algorithms as smooth and seamless as possible, while still allowing flexibility in customization. The basis and flow of the language was inspired by looking at pseudocode within CLRS, and trying to implement them as closely as possible in a syntax similar to the C++ programming language.

2 Data Types

Type	Description
int	32-bit integer
double	Double precision floating point
string	Text
bool	Boolean
graph<T1, T2>	Graph with of type T1 and edge weight type T2
node<T>	Node of type T
array<T, n>	Array of type T with size n
list<T>	Linked list of type T
tuple<>	Tuple with heterogeneous elements

3 Keywords

Keywords	Description
if, else if, else	Conditional statements
for, while	For-loops, while-loops
return	Exit function and return value
continue	Skip to the next iteration of the loop
break	Exist the loop
<type> <name> (<parameters>) {}	Function declaration
;	End of statement
{}	Initialization list, scope declaration

4 Operators

Operations	Description
=	Assignment
+, -, /, *	Integer/double arithmetic
==, <, <=, >, >=, !=	Integer/double comparison
||, &&, !	Logical operators
-(w)>,	Create a directed edge with weight w between two nodes
->	Create a directed edge with weight 1 between two nodes
-(w)-	Create an undirected edge with weight w between two nodes
--	Create an undirected edge with weight 1 between two nodes

5 Built-in Functions

Built-in Functions	Description
node<T> n {k, v}	Create a node with a key of type int and a value of type T
n.edges()	Return an array containing tuples {weight, node pointed to, traversable} for the edges of a node
g.add(node<T> n)	Add a node to a graph; Can create node within the function
g[k]	Access a node within a graph by its key
g.del(k)	Delete a node within a graph by its key
a[i]	Access an element in an array by its index
l.push_front(T e)	Add an element of type T to the front of a list
l.push_back(T e)	Add an element of type T to the back of a list
l.pop_front()	Remove and return the element at the front of the list
l.push_back()	Remove and return the element at the back of the list

6 Sample Code

```
// Declare a graph
graph <int> g;

# Adds nodes to the graph
for(int i = 0; i < 10; i++){
   g.add(i,2*i)
}

// Create an edge of weight i from root node to every other node
for(int i = 1; i < 10; i++){
   g[0] -(i)> g[i]
}

// Overwrite edge from g[0] to g[2] with weight 10
g[0] -(10)> g[2]

// BFS search example (Takes in graph and destination)
bfs(graph<type> g, node<type> n){

   list<type> q;
   graph<type> d;

   q.push_back(g.root());
   while(!q.empty()){
       node<type> m = q.pop_front();

       if(m == n){
           return m;
       }

       for(node<type> e : v.edges()){
           if(!discovered.contains(v)){
               q.push_back(e)
           }
       }
   }
}
```