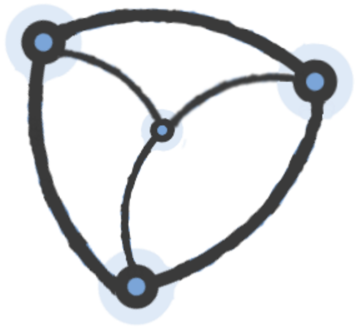

Graphene

Project Proposal

Adith Tekur (at2904), Neha Rastogi (nr2477), Sarah Panda (sp3206)
Sumiran Shah (srs2222), Pooja Prakash(pk2451)
Team 8

February 26, 2014

1 Introduction

The motivation for our language is the massive commonplace use of graphs and graph based data mining algorithms in today's software world. At the same time, we see a large bubble of social network and social network-like applications which manage a large backend of data which can usually be represented using a graph structure. Most of today's languages do not provide out-of-the-box or easy to use features for graph initialization, operations and management. Our language aims to provide this interface to be able to support generic graph algorithms as well as specific social network applications based computations on graph-like data structures.

The target audience for Graphene is twofold. First, beginner programmers who have little background in implementing and managing data structures in detail but wish to work on graphs. Second, organizations holding massive data in graph formats and who need to have dedicated computational services to model and mine data from graphs. Our language can be used to both, model new social networks as well as import data from existing dumps of social networks such as facebook, linkedin, twitter and compute on them.

Buzzwords: High level, interpreted, easy to use, powerful, concise, minimal development time, social networks, graph

2 Language basics

2.1 Primitive data types

Primitive data types which are specific to Graphene are:

1. Graph: A graph represents a collection of nodes interconnected by edges. A graph can be directed or undirected depending on whether the direction of edges have meaning.
2. Node: A node is an element amongst a set of similar elements of a graph. Each node has a few implicit language defined properties. User defined properties can be added as key-value pairs in an associative property array.
3. Edge: An edge connects two nodes of a graph. Similar to nodes, edges have a few implicit properties and more can be defined by key-value pairs in the associative property array.

Standard data types including String, int, float, double, char, boolean and arrays of all types are also supported by our language.

2.2 Declaring a graph

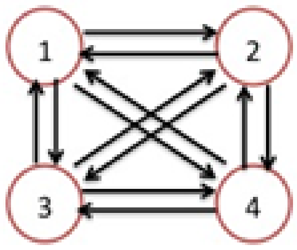

Figure 1: An example of a complete mesh graph.

Since a graph can be either directed or undirected, Graphene provides users with a type-specifier to indicate the same. *d* and *u* represent directed or undirected graphs respectively.

1. Method 1

   ```
   Graph g = d[1 <-> 2, 1 <-> 3, 1 <-> 4, 2 <-> 3, 2 <-> 4, 3 <-> 4];
   ```

   Here we list all edges of the graph. Two directional edges pointing in the opposite directions, or edges for undirected graphs are denoted by a `<->` operator. A single directional edge can be specified by a `->` operator. However, it may be easier to sometimes specify the edges *absent* from a graph which would otherwise be a full mesh network. For this, the user has can make use of method 2 as listed below.

2. Method 2

   `Graph g = Graph.mesh(` *nodecount* `,` *type* `);`

   ```
   e.g.  Graph g = Graph.mesh(4, d); // This generates a graph as in Figure 1
   g = g - (g.["1"] -> g.["2"]);
   ```

   Graph.mesh(int n) returns a full mesh graph containing *n* nodes. It automatically allocates unique IDs for the nodes ranging from 1 to *n*. The code snippet creates a graph with 4 nodes with all possible edges except the edge pointing from node 1 to node 2.

3. Method 3

   ```
   Graph g = d[1[2[1,3[1,2,4[1,2,3]],4],3,4]];
   ```

   We begin by enumerating the IDs of all the nodes. At the first occurance of each node, we expand it to include all its neighbors to which its edges point. This method is easy to use when a user wants to define sparse graphs. However, for defining dense graphs, the user is recommended to use methods 1 or 2.

2.3 Properties of Nodes

Each node has a unique node ID. A node can belong to multiple graphs. Hence, a node has a list of graph IDs to which it belongs to. A node in a graph may represent anything that can be modeled as a node in a graph. For example, in the domain of social network graphs, user accounts and pages can be represented as nodes. Hence, we allow for a user-defined list of properties through the *property* parameter. For example, a user may specify "name" as a property of the node. This can be done as *property*["*name*"]. Assigning a value to this can be done using the = operator as *property*["*name*"] = <*value*>.

2.4 Properties of Edges

Similar to a node, each edge has a unique edge ID, the graph ID to which it belongs to and a user defined list of properties *property*. The *property* attribute of an edge determines what the edge between two nodes signifies. It is this attribute that defines the relation between two nodes. If the two nodes connected by an edge represent user accounts, the edge may represent a relation of friendship between the them or a relation of having similar interests. The *property* atribute gives the user the freedom to define such relations.

2.5 Syntax

2.5.1 Comments

Comments supported Graphene are the typical single-line and multi-line block.

```
// This is a single line comment.

/*
    This is a
    multi-line
    comment.
*/
```

2.5.2 Operators

We provide the basic arithmetic and logical operators. Other graph specific operators are:

1. Edge operators '<->', '->'
2. Node addition operator '+'

   ```
   graphObj = graphObj + graphObj.["1"];
   ```

3. Edge addition operator '+'

   ```
   graphObj = graphObj + (graphObj.["2"] -> graphObj.["name":"John"]);
   graphObj = graphObj + (graphObj.["1"] <-> graphObj.["name":"Bob"]);
   ```

4. Node removal operator

```
graphObj = graphObj - graphObj.["name":"Alice"];
```

5. Edge removal operator

```
graphObj = graphObj - (graphObj.["2"] -> graphObj.["name":"John"]);
graphObj = graphObj - (graphObj.["1"] <-> graphObj.["name":"Bob"]);
```

2.5.3 Control Flow

```
if (bool)
{
    // matched first try!
}
else if (otherBool)
{
    // that did not match, this did!
}
else
{
    // nothing matched
}

// foreach loops
foreach(Node n in Graph g)
{
}

foreach(Edge n in Graph g)
{
}

// for loops
for (i = 0; i < 10; i++)
{
}

// while loops
while (bool)
{
}
```

3 Standard Library Functions

The use of inbuilt library functions provide an intuitive sense of how to deal with the graph data type. *this* refers to the object that the function was called with.

3.1 Shorthand notation

`["":""]` is the supported way to handle stored key-value data.
By default, `[""]` refers to value stored for key ID.

```
graphObj.["":""];
graphObj.[""]; // returns Node

nodeObj.["":""];
nodeObj.[""]; // returns value for key, default is ID
```

3.2 General graph methods

```
graphObj.isDirected(); // returns boolean

graphObj.getNode(["":""]).listAdjacent(); // returns list
graphObj.getNode(["":""]).listNonAdjacent(); // returns list

graphObj.hasNode(["":""]); // returns boolean
graphObj.hasNode([""]); // default key is ID

graphObj.hasEdge(["":""] -> ["":""]); // returns boolean
graphObj.hasEdge([""] <-> [""]); // default key is ID
```

3.3 Node Handling:

```
graphObj.getNode(["":""]); // returns Node
graphObj.getNode([""]);

graphObj.addNode(["":"", "":"" , "":""]); // returns reference to newly created Node
// If ID is not specified, next available ID is allocated

graphObj.deleteNode(["":""]); // returns boolean, true on success.

nodeObj.connectTo(Node); // returns boolean, true on success.
nodeObj.disconnectFrom(Node); // returns boolean, true on success.
```

3.4 Node Degree calculations:

```
graphObj.getNode(["":""]).inDegree(); // returns int
graphObj.getNode(["":""]).outDegree(); // returns int
graphObj.getNode(["":""]).degree(); // returns int
```

Note: Be cautious while calculating degrees for undirected graphs. degree != indegree + outdegree.

3.5 Edge & Node count:

```
graphObj.listEdges().count();
graphObj.listNodes().count();
```

3.6 Path between two nodes:

```
graphObj.getPath(["":""],["":""]); // returns list Node objects constituting the path
graphObj.getPath(["":""],["":""]).count() -1; // returns hop count
```

3.7 Tree Generation

DFS Tree

`graphObj.getDFS(Node root,` *lambda expr*`); // returns graph which is a DFS tree`
lambda is optional. Default behaviour is to prefer lower node IDs to the left.

```
        e.g. graphObj.getDFS(this.[""] , lambda resolve(Node n1, Node n2) :
                                    if(n1.["id"] > n2.["id"]) n1 else n2);
```

BFS Tree

`graphObj.getBFS(Node root,` *lambda expr*`); // returns graph which is a BFS tree`
lambda is optional. Default is lesser node IDs to the left.

```
        e.g. graph.getBFS(this.[""] , lambda resolve(Node n1, Node n2) :
                                 if(n1.["id"] > n2.["id"]) n1 else n2);
```

3.8 Clustering based on user-specified lambda

```
graphObj.cluster(lambda expr); // returns graph[]
```

lambda expr is of signature *boolean() (Node n1, Node n2)* which returns true if $n1$ and $n2$ are tested to belong to the same cluster.

```
        e.g. graphObj.cluster(lambda node: graph.hasEdge(node <-> [""]));
```

3.9 Merging

Merging two nodes simply requires specifying the two nodes.

```
node.mergeWith(graphObj.getNode(["":""]));
node.mergeWith(graphObj.getNode([""]));
```

Merging two graphs on the other hand requires two lambda functions, one to test for mergability and the second to specify how the resultant node is generated.

```
graphObj.mergeWith(graph, lambda expr: fn(Node n1, Node n2), lambda expr: resultNode())
        e.g. graphObj.mergeWith(graphObj, lambda mergable(Node n1, Node n2) : \
                result = (n1.["id"] == n2.["id"]) , lambda resultNode(Node n1, Node n2) : \
                { result.["id"] = n1.["id"] ; result.["name"] = n1.["name"] +" "+ n2.["name"] ; });
```

3.10 Constraints & Groups

Constraints can be specified in terms of lambda functions. These constraints are checked at every modification to the graph structure henceforth. Constraints can be, for example, in the form of "two specific nodes cannot have a common edge" or be computed by the lambda expression. If the lambda expression evaluates to be true, the constraint is enforced.

```
graphObj.addConstraint(String constraintName, lambda expr);
graphObj.removeConstraint(String constraintName);
```

Groups (list of nodes; user-defined)

```
groupObj.fromGraph(graphObj, lambda expr).fromGraph(graphObj, lambda expr);
// returns list of grouped nodes
```

4 Sample Programs

4.1 Scenario: Create a network, identify all the closely-knit groups and find who is the go-to person in each.

```
Nodes[] findMostInfluential(Graph G)
{
    // By default it returns the strongly connected component
    Graph[] Subgraphs = Cluster(G, lambda membership(Node n1, Node n2) : this.hasEdge(n1, n2));
    int maxdegree, degree;
    Node[] Minfluential = new Node[];
    Node influential;
    foreach(graph in Subgraphs)
    {
        maxdegree=0;
        influential = graph[0];
        foreach(Node user in Graph graph)
        {
            degree = indegree(graph,user)
            if maxdegree < degree
            {
                maxdegree = degree;
                influential = user;
            }
         }
         Minfluential.add(influential);
    }
    return Minfluential;
}

main()
{
    Graph g = d[1->2,3->5,2->5,3-4];
    Nodes[] gotoperson = findMostInfluential (g);
    display "Found "+ gotoperson.length() + "Closely-Knit groups";
    plot(g, highlight(gotoperson));
}
```

4.2 Scenario: Given a node representing a url, generate a graph of other webpage nodes which carry a common characteristic (for example websites that allow google adsense), that is defined by a lamda function.

```
Graph SearchBot(url, lambda){
  Graph connections = new Graph();
  connections = connections + connections.["url":url];
Queue url_q =  new Queue();
  url_q.push(url);
  while (url_q.length()!=0){
    c_url = url_q.pop();

    // user defined function to parse the contents of the page and extract the links
    links = getlinksfromHTMLcontents(c_url);

    foreach (link in links){
        if(lambda(url)==True){
          url_q.push(link);
          connections = connections + connections.[link];
          connections = connections + (connections.[c_url] -> connections.["url":link]);
        }
    }
  }
 return connections;
}
```

4.3 Scenario : Given two different social network graphs and a user who is new to one of the networks, build a recommendation system that tells the user who all on this new network he might now, by analysis of overlap of the two networks.

```
Node[] WhoYouMayKnow(Graph OldGraph, Graph NewGraph, Node user){
  Node[] OldFriends = user.listAdjacent();
  Node[] Suggestions = new Node[];
  foreach (friend in OldFriends){
  if (NewGraph.hasNode[""] == True & NewGraph.hasEdge([] -> [friend.["id"]] == False))
      Suggestions.add(friend);
  }
  return Suggestions;
}
```