GRI: Interpreter of a dynamic language for GRaph algorithms

Sandeep Dasgupta
University Of Illinois at Urbana Champaign.
sdasgup3@illinois.edu

ABSTRACT

As graphical models are increasingly become popular in various field, the domain experts often struggle to represent and compute on such model in a convenient and efficient way. In this project we develop a dynamically typed language to represent and compute on the graphical models which provides both the desired convenience without loosing much on the efficiency.

Keywords

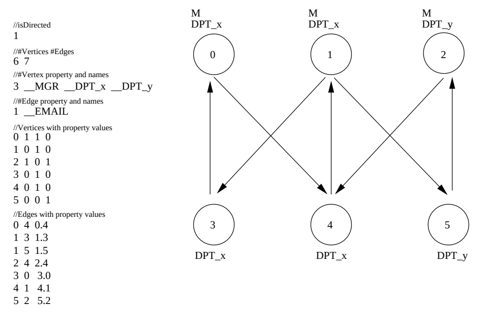
Graph Algorithm, dynamically typed language, interpreter

1. INTRODUCTION

As graphical models are increasingly being used in various fields like biochemistry (genomics), electrical engineering (communication networks and coding theory), computer science (algorithms and computation) and operations research (scheduling), organizational structures, social networking, there is a need to represent and allow computation on them in a convenient and efficient way. This involves (but not limited to)

- Designing a language which provide an convenient interface to the programmer to program those models. This is essential so that even for domain experts who are not coding experts can code and reason about their implementation. Ease of interface could be due to:
  - Expressive power of the language representing those models.
  - Intuitive extensibility of the language.
  - Ability of the language to provide exploratory programming, where the user may experiment with different ideas (without dwelling much into the language syntax) before coming to a conclusive one.
- Designed language need to be efficient in the following sense.
  - Underlying design decisions including data structures need to be carefully crafted to achieve expected runtime w.r.t the input size.

.



```
//isDirected
1
//#Vertices #Edges
6  7
//#Vertex property and names
3  __MGR  __DPT_x  __DPT_y
//#Edge property and names
1  __EMAIL
//Vertices with property values
0  1  1  0
1  0  1  0
2  1  0  1
3  0  1  0
4  0  1  0
5  0  0  1
//Edges with property values
0  4  0.4
1  3  1.3
1  5  1.5
2  4  2.4
3  0   3.0
4  1   4.1
5  2   5.2
```

  - Implementation need to be scalable w.r.t the space/time requirements. This is important because most of the graph algorithm typically work on huge input sizes.

2. RELATED WORK

Our work in mostly inspired by the line of work by GUESS [**?**] and Graphal [**?**]. GUESS, a novel system for graph exploration that combines an interpreted language with a graphical front end that allows researchers to rapidly prototype and deploy new visualizations. GUESS also contains a novel, interactive interpreter that connects the language and interface in a way that facilities exploratory visualization tasks. Our language, Gython, is a domain-specific embedded language which provides all the advantages of Python with new, graph specific operators, primitives, and shortcuts.

Graphal is an interpreter of a programming language that is mainly oriented to graph algorithms. There is a command line interpreter and a graphical integrated development environment. The IDE contains text editor for programmers, compilation and script output, advanced debugger and visualization window. The progress of the interpreted and debugged graph algorithm can be displayed in 3D scene.

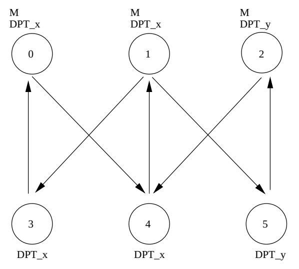
Our language design is very similar to above two work. But we additionally provided built-in functions for basic graph algorithms. This not only help us getting convinient short hand notations to compute those basic algorithms, but also we gain on performance due the fact that those basic algorithms are now compiled.

3. A MOTIVATING EXAMPLE

In this section we will provide some insight of designed language using some motivating examples.

EXAMPLE 1. *In the following figure we have a directed graph where the nodes represent the employees and the edges between them represents the mail conversation from one employee to other. For example,* `Node 0` *represemts an employye in* `DPT_x` *who is a*

```
function main(argv) {

    g = graph();
    g.setDirected(true);

    v0 = g.createVertex();
    v0.__id = 0;
    v0.__DPT_x = 1.0;
    v0.__DPT_y = 0.0;
    v0.__MGR    = 1.0;

    v3 = g.createVertex();
    v3.__id = 3;
    v3.__DPT_x    = 1.0;
    v3.__DPT_y   =  0.0;
    v3.__MGR  =     0.0;

    g.createEdge(v3, v0);

}
```



```
edge.getBeginVertex();
edge.getEndVErtex();
g.getVertices();
g.getEdges();
```

```
         function main(argv) main() {
  S1:      g = graph();
  S2:      ...
  S3:      g.setDirected(true);
  S4:      ...
  S5:      v0 = g.createVertex();
  S6:      v0.__id = 0;
  S7:      v0.__DPT_x = 1;
  S8:      v0.__DPT_y = 0;
  S9:      v0.__MGR = 1;
 S10:      ...
 S11:      v3 = g.createVertex();
 S12:      v3.__id = 3;
 S13:      v3.__DPT_x = 1;
 S14:      v3.__DPT_y = 0;
 S15:      v3.__MGR = 0;
 S16:      ...
 S17:      g.createEdge(v0,v3);
         }
```

Manager as well (this is attributed by the `MGR` *tag). Similarly,* `Node 3` *represents a* `DPT_x` *employee. The edge between* `Node 3` *and* `Node 0` *represents the email conversation from employee* `Node 3` *to employee node* `Node 0`.

Now lets first talk about the ways to represent this graph. One way is as shown in fig. At line `S1`*, a new graph variable is created.* `S3` *sets that the graph is directed.* `S5` *create a node* `Node 0` *of this graph. As nodes and edges are the basic building block of a graph we have made them the frst class objects which allows users to access them directly. Lines* `S6 - S9` *sets various properties of the vertex. For example, it says that the vertex has* `__id = 0` *, it belongs to* `__DPT_x`*, does not belongs to* `__DPT_y` *and its* `__MGR` *property is true. Similar properties are set for vertex node with* `__id = 3`*. And finally a directed edge between them is created at line* `S17`.

One thing to note here is that the methods like `graph`, `setDirected`, `createVertex` *and* `createEdge` *are all built-in compiled functions.*

Figure shows another way to represent graph. It uses an input file to be fed to the interpreted program. The format of the input file is shon in Figure. Line `S2` *loads the input file using command line arguments. At*
t S3, we can see two built-in functions `getAdjacencyMatrix` *&* `getVertices` *on graph which respectively gives an array of array representing the adjacency matrix representation of the graph and a set of vertices in the graph.* `displayAdjMatrix` *is just a method call, the definition of which is not shown for brevity. Now lets try to solve certain queries from the graph at Fogure. In case we want to get all the nodes in the graph who are* `__DPT_x` *employees, then*

```
         function main(argv) main() {
  S1:      g = graph();
  S2:      g.loadFromFile(argv[0]);
  S3:      displayAdjMatrix(g.getAdjacencyMatrix(), g.getVertices());
  S4:      ...
  S5:      dpt_x_employee = g.getVertexSetWithProperty("__DPT_x", 1.0);
  S6:      mgr_employee = g.getVertexSetWithProperty("__MGR", 1.0);
  S7:      ...
  S8:      /* Set of __DPT_x employees who are __MGR as well */
  S9:      dpt_x_AND_mgr = dpt_x_employee.intersection(mgr_employee);
 S10:      ...
 S11:      println("Set of __DPT_x employees who are __MGR as well");
 S12:      foreach(employee; dpt_x_AND_mgr) {
 S13:        println(" " + employee.__id + " ");
 S14:      }
 S15:      ...
 S16:      /* Set of __DPT_x employees who are NOT __MGR */
 S17:      dpt_x_MINUS_mgr = dpt_x_employee.difference(mgr_employee);
 S18:      ...
 S19:      it = dpt_x_MINUS_mgr.iterator();
 S20:      ...
 S21:      println("Set of __DPT_x employees who are NOT __MGR");
 S22:      while(it.hasNext()) {
 S23:        employee = it.next();
 S24:        println(" " + employee.__id + " ");
 S25:      }
 S26:      ...
 S27:      // Email Exchanges from MGRs to non-MGR DPT_x employee
 S28:      emailExchanges = mgr_employee -> dpt_x_MINUS_mgr;
 S29:      ...
 S30:      // Email Exchanges from non-MGR DPT_x employee to MGRs*/
 S31:      emailExchanges = mgr_employee <- dpt_x_MINUS_mgr;
 S32:      ...
 S33:      // Email Exchanges between non-MGR DPT_x employee and MGRs*/
 S34:      emailExchanges = mgr_employee <-> dpt_x_MINUS_mgr;
 S35:      ...
         }
```

the query to get all those nodes is at line `S5`. *Similarly, the query at line* `S6`, *gives the nodes for wehich the property* `__MGR` *is set to true. One thing to note here that the return value of both the queries are sets which are amenable tos et operations.*

For example, in cse we want to get all the employees who are both depratment `__DPT_x` *empoyee and managers , we can get that using the set intersection as shown at line* `S9`. *Line* `S8`, *shows the multilien comment we support. Line* `S9` *shows a way to iterate over the set elements using foreach construct.*

Again, if we want to get the set of __DPT_x employees who are ***NOT*** `__MGR`, *we can write a query like at line* `S17`. *Line* `S19` *shows another ways to get an iteration to itearte on composite data structures.*

Now once we have two sets of nodes each with a specific property, we can query for the edges between them. For example, lines `S28`, `31` *and* `34` *gives the set of edges emanating from one set of nodes to onother set of nodes. Note that operators* ← *and* → *are applicable to undirected graphs and the operator* ↔ *is applicable to both directed and undirected graphs. For directed graphs, it gives all the bidirectional edges between two node sets and for undirected graphs, it returns all the edges between two node sets.*

□

EXAMPLE 2. *Let consider another example implementation the depth first traversal on a graph.*

```
define("NUM_VERTICES", "10");
define("NOTVISITED", "0");
define("VISITED", "1");

function dfs(v, dfsorder)
{
    if(v.visit == VISITED)
      return;

    println("vertex visited: " + v.num);
    v.visit = VISITED;

    dfsorder.pushBack(v);

    foreach(neighbor; v.getNeighbors())
      dfs(neighbor, dfsorder);
}

function main(argv)
{
  g = graph();
  g.loadFromFile(argv[0]);

  dfsorder = array(0);
  dfs(first, dfsorder);

  for(vertex: dfsorder ) {
    println(" " + vertex.__id + " " );
  }
}
```

□

4. DEFINITIONS AND NOTATIONS

4.1 Interprocedural Analysis