GRI: **I**nterpreter of a dynamic language for **GR**aph algorithms

Sandeep Dasgupta

1st December 2015

Outline

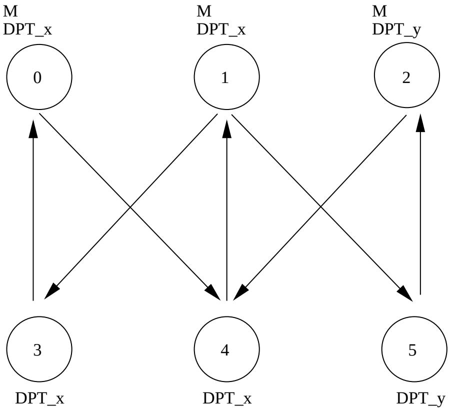
Motivation

- Graphical models are applied to widely varying fields.
  - Biochemistry
  - Electrical Engineering
  - Computer Science
  - Operations Research
  - Organizational Structures

- To represent & allow computations on graphical models in **Convenient** and **Efficient** way.

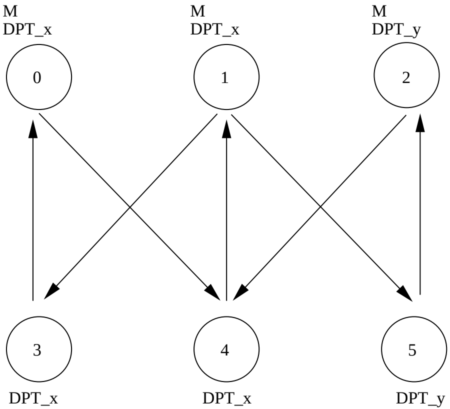
Outline

```
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);

}
```

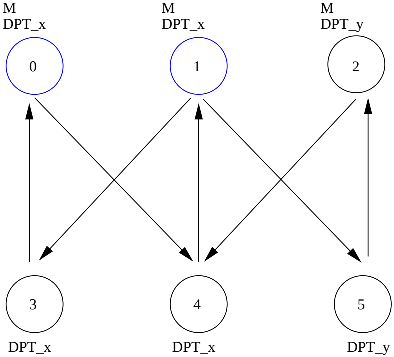



```
edge.getBeginVertex();
edge.getEndVErtex();
g.getVertices();
g.getEdges();
```

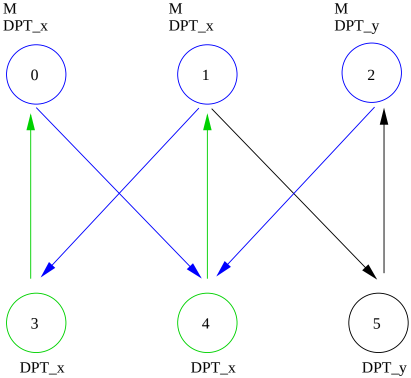
```
//isDirected
1
//#Vertices #Edges
6 7
//#Vertex property and names
3 __MGR __DPT_x __DPT_y
//#Edge property and names
1 __EMAIL
//Vertices with property values
0 1 1 0
1 0 1 0
2 1 0 1
3 0 1 0
4 0 1 0
5 0 0 1
//Edges with property values
0 4 0.4
1 3 1.3
1 5 1.5
2 4 2.4
3 0  3.0
4 1  4.1
5 2  5.2
```



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

    matrix = g.getAdjMatrix();

    dpt_x_employee = g.getVertexWithProperty("__DPT_x", 1.0);
                                                    //Gives:  {0, 1, 3, 4}

    mgr_employee  =  g.getVertexWithProperty("__MGR", 1.0);
                                                    //Gives: {0, 1, 2}

    dpt_x_AND_mgr = dpt_x_employee.intersection(mgr_employee);
                                                    //Gives: {0,1}

    println("Set of __DPT_x employee who are __MGR as well");

    foreach(employee: dpt_x_AND_mgr) {
      println(" " + employee.__id + " ");
    }
}
```



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

    matrix = g.getAdjMatrix();

    dpt_x_employee = g.getVertexWithProperty("__DPT_x", 1.0);
                                                        //Gives:  {0, 1, 3, 4}

    mgr_employee  =  g.getVertexWithProperty("__MGR", 1.0);
                                                        //Gives: {0, 1, 2}


    /* Set of __DPT_x employees who are not __MGRs*/
    dpt_x_MINUS_mgr = dpt_x_employee.difference(mgr_employee);
                                                                  //Gives: {3, 4}

    it  = dpt_x_MINUS_mgr.iterator();
    println("Set of __DPT_x employee who are not __MGRs");
    while(it.hasNext()) {
        employee = it.next();
        println(" " + employee.__id + " ");
    }

     emailExchanges = mgr_employee -> dpt_x_MINUS_mgr;
                                          //Gives: (0, 4), (1,3), (2,4)

     emailExchanges = mgr_employee <- dpt_x_MINUS_mgr;
                                                   //Gives: (3,0), (4,1)

     emailExchanges = mgr_employee <-> dpt_x_MINUS_mgr;
                               //Gives: (0,4), (1,3), (2,4), (3,0), 4,1)
 }
```



```
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 + " " );
  }
}
```

Outline

- Interprets the AST.
- AST structure.
    - AST : list of function definitions.
    - Leafs could be `identifier, int, float, true, false, null, string, vertex, edge or graph.`
- How Runtime works.

Outline

Naive Implementation

Table: Slowdown of our implementation w.r.t C implementation. The graph used in all the cases consist of vertices = 500, edges = 1000. For Graph Coloring, we used the graph with vertices = 125, edges = 1000.

Algorithm	Fully Scripted time (secs)	C implementation time(secs)	Slowdown
Transitive Closure (Floyd Warshall)	1137.40	4.04	281.5
Shortest Path (Dijkstra)	6.37	0.02	318.5
Minimum Spanning Tree (Prim)	6.34	0.02	317.0
Graph Coloring (Chaitin Optimistic)	138.17	0.65	212.6

Built-in Functions

- `Graph::getAdjMatrix();`
- `Graph::getTransitiveClosure();`
- `Graph::getShortestPath(string wt, Vertex start, vertex end);`
  - Returns the shortest distance of end from start.
  - Returns parent of each vertex in the shortest path tree.
  - If end == NULL, return the shortespath from start to all vertices.
  - If end != NULL, return the shortespath from start to end.
- `Graph::getMST(string wt);`
  - Return the parent of each vertex in the minimum snapping tree.

Built-in Functions

Table: Speedup of the our implementation with built-in functions w.r.t without using built-in functions. The graph used in all the cases consist of vertices = 125, edges = 1000

	With Built-ins time (secs)	Fully Scripted time(secs)	Speedup
Transitive Closure (Floyd Warshall)	0.54	18.19	33.6
Shortest Path (Dijkstra)	0.08	0.51	6.3
Minimum Spanning Tree (Prim)	0.05	0.47	9.4

Revised Implementation

Table: Slowdown of our implementation w.r.t C implementation. The graph used in all the cases consist of vertices = 500, edges = 1000. For Graph Coloring, we used the graph with vertices = 125, edges = 1000.

	GRI Time(secs)	C Time(secs)	Slowdown
Transitive Closure (Floyd Warshall)	10.26	4.04	2.5
Shortest Path (Dijkstra)	0.09	0.02	4.5
Minimum Spanning Tree (Prim)	0.09	0.02	4.5
Graph Coloring (Chaitin Optimistic)	138.17	0.65	212.6

Outline

- Comparison with an existing implementation.
- To support saving of state and retrieving it back.
- To add relevant built-in functions to build complex algorithms and improve efficiency.