Progress Report: Designing an Interpreter for a dynamic language for graph algorithms

Sandeep Dasgupta*

November 15, 2015

*Electronic address: sdasgup3@illinois.edu

1 Problem Statement

In this project we are planning to work on a dynamically typed language to represent graphs and apply various computations on them.

2 Language Syntax

The syntax of the language is an oversimplified version of C, but without the mention of any types. The operations on incompatible types will be error-ed out while interpreting.

2.1 Progress

- We have implemented the tokenizer using flex. Appendix A shows the tokens sent to the parser routine.
- We are supporting syntax like #include("filename") and #define("PI", "3.14") while doing a single pass of parsing (i.e. both preprocessing of these constructs done while parsing). This is achieved by using flex internal stack to manage multiple buffers.
- Appendix B shows the parser rules. These are borrowed from `http://www.quut.com/c/ANSI-C-grammar-y.html`. The rules are compiled by bison tool to generate the C parser.
- We are done with generating the AST. Our AST is basically a list of function definitions. Each function definition class holds name of the function, a set of formal arguments and a list of body statements. These body statements could be a assignment, loop-statement, function call, etc. The leafs of the AST could be an identifier, int, float, true, false, null, string, vertex, edge or graph.
- Some of the key features of the parser is as follows:
  - Support of C statements like *if then*, *if then else*, *while*, *for*.
  - Support of break, continue within loop constructs and return in function body. As we are representing both loop-body and function body (i.e. anything between "{" & "}") as compound statements so we do not have to distinguish the two cases. The semantics of executing a break, continue and return will be discussed in the semantics section.
  - Supporting vertices, edges and graphs as first class objects valuevertex, valueedge, valuegraph respectively. The syntax to declare a graph is `g = graph();` which will be parsed as a assignment node with left Value as an identifier and right value as a function call. Now this function call corresponds to a built in function that returns a valuegraph (which is of one the leaf nodes of AST).
  - The first class object of vertex and edge contains a map to add properties. This feature is useful in various graph algorithms like in dfs traversal we may use a vertex property "visited" to keep track of vertices already explored.

3 Language Semantics

- As "nodes" and "edges" are the backbone of any graph structure, we proposed to have them as first-class objects. Even though the user will not necessarily be accessing individual nodes/edges, but the option is available to them.

- Providing users with the option to specify attributes on nodes and edges and later use those attributes for computing results. For example: to find all the red colored nodes (where color is marked as an attribute of node) connected to a specific node.

4 Interpretor Runtime

5 Future Work

The following are the future work.

- To support additional operators to specify relationships between the nodes/edges or their groups. The operator that we are planning to support are the described next with their semantics.
  - v1←v2: Select all the directed edges from vertices v1 to v2.
  - v1→v2: Select all the directed edges from vertices v2 to v1.
  - v1↔v2: Select all the directed edges between vertices v1 and v1.
  - v1?v2: Select all the edges (directed or undirected) between vertices v1 and v1.
  - S1∩S2: Selects the intersection between the set of vertices/edges.

  Most of these ideas behind choosing the operators are borrowed from [2]. The intuition behind these operators is that the users do not have to remember longer commands. Also its very intuitive to build up complex operations using the simpler ones.
- To support saving of state and retrieving it back using routines like saveSateFromFile & loadStaeFromFile. This idea is borrowed from [2] and this seems a useful service provided to do exploratory programming, as the user might be interested to checkout the last saved state (or a state with a any tag) or to undo all the experiments down to a particular state.

5.1 Evaluation Strategy

The baseline of our evaluation will be the open source [1] system. The evaluation will cover the following two aspects of our implementation:

- The convenience and intuitive extensibility provided by our programming model.
  - We will be implementing a couple of well know graph algorithms in both baseline and our implementation and use number of dynamic instructions interpreted as a measure to show the conciseness of our representation.
- Performance
  - As we are planning to keep the compiled version of frequently used graph routines (like dfs_iterators(vertex), bfs_iterators()), we are expecting to achieve better performance in terms of runtime.

References

[1] *Graphal: Graph algorithms interpreter.*

[2] E. Adar, *Guess: A language and interface for graph exploration*, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '06, New York, NY, USA, 2006, ACM, pp. 791–800.