Designing an Interpreter for a dynamic language for graph algorithms

Sandeep Dasgupta*

October 30, 2015

*Electronic address: sdasgup3@illinois.edu

1 Motivation

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 run-time w.r.t the input size.
  - 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.

To meet all above goals and most importantly exploratory programming, we decided to work on a dynamically typed language to represent graphs and apply various computations on them. With a dynamically typed language the user do not have to worry much about declaring types and can focus mostly on his/her experiments.

Following is the outline of the proposed features of the language. These are written keeping the "convenience" aspect of the language in mind.

- 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.
- Supporting operators to specify relationships between the nodes/edges or their groups. For example, the user can create sub graphs like

$$department\_1 = (N1, N2, N3) \tag{1}$$

$$managers = (N3, N4, N5) \tag{2}$$

  and use operators like $\cap$ to find nodes "department 1 employees who are managers as well". Also the language supports operators to find edges between groups of nodes. This will answer interesting queries like:

$$managers \rightarrow (department1 - managers)$$

  could mean "the number of email conversation between managers to department_1's non managers"

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.

- Also borrowed from [2], is the option of saving state of the graph. 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.

In this project we will also be implementing an Interpreter of the above mentioned language. We call it GRI (Graph Interpreter).

2 GRI Interpreter

Proposed language will not allow the user to specify the types and a variable may point to objects of different types during run time (and hence a dynamically types language). In this scenario we could do a static compilation after a static type inference to generate sub optimal machine code, whose efficiency depends on how well the types are inferred.

As opposed to that, we are trying to build an interpreter which will interpret the AST spit out by the parser. If time permits, we are planning to convert the the AST into LLVM byte code (after static type inferencing) to be Jitted by the LLVM tool-chain.

3 Language

The syntax of the language is going to be an oversimplified version of C. Also planning to provide a pre-compiled library of rich set of built-in functions like *deleteVertex(graph, vertex), dfs_iterators(vertex), bfs_iterators(vertex)* which will facilitate writing complex algorithms conveniently and enhance the execution time of interpreting. Some of the ideas of the language design will be borrowed from [1].

The language will distinguish between user variables/functions and system variables/functions. The later will be treated immutable and an error will be issued if the user tried to modify this. For example, a system variable g, representing a graph, cannot to modified (for example, by assigning g = 5;). In that sense the "dynamically typed" nature of the language will be restricted to the user variables.

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.